Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997

Parameterized Types for Java

Andrew C. Myers Joseph A. Bank Barbara Liskov

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139
{andru,jbank,liskov}@lcs.mit.edu

Abstract

Java offers the real possibility that most programs can be written in a type-safe language. However, for Java to be broadly useful, it needs additional expressive power. This paper extends Java in one area where more power is needed: support for parametric polymorphism, which allows the definition and implementation of generic abstractions. We discuss both the rationale for our design decisions and the impact of the extension on other parts of Java, including arrays and the class library. We also describe optional extensions to the Java virtual machine to allow parameterized bytecodes, and how to verify them efficiently. We have extended the Java bytecode interpreter to provide good performance for parameterized code in both execution speed and code size, without slowing down non-parameterized code.

1 Introduction

Java [Sun95a] is a type-safe, object-oriented programming language that is interesting because of its potential for WWW applications. Because of the widespread interest in Java, its similarity to C and C++, and its industrial support, many organizations may make Java their language of choice.

Java is also interesting because of its machine-independent target architecture, the Java Virtual Machine (JVM) [LY96]. Java is attractive for web applications, in part because Java bytecodes can be statically *verified*, preventing violations of type safety that might access private information.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, and in part by a grant from Sun Microsystems.

Java as currently defined is explicitly a first version that is intended to be extended later. This paper addresses one of the areas where extension is needed, namely, support for generic code. In Java, it is possible to define a new type, such as a set of integers, but it is not possible to capture a set abstraction where the elements of a particular set are homogeneous, but the element type can differ from one set to another. Current Java programs adapt to the lack of genericity by using runtime type discrimination, which is awkward for the programmer and also relatively expensive.

In this paper, we extend Java with parametric polymorphism, a mechanism for writing generic interfaces and implementations. We provide a complete design of this extension and discuss the interaction of parametric polymorphism with other Java features, including ways that generic abstractions can simplify coding with some standard Java classes. We also show how to implement this design efficiently. We sketch an implementation of the design using the existing JVM bytecodes, and also present an extension to the JVM that provides better performance. Finally, we present results from a working prototype of the extended JVM interpreter, showing that our technique improves performance by up to 17% over standard Java. Detailed specifications of our extensions are available in the appendices.

An explicit goal of our work was to be very conservative. We extended Java by adapting an existing, working mechanism with as few changes as possible. We supported the Java philosophy of providing separate compilation with complete inter-module type checking, which also seemed important for pragmatic reasons.

We used the Theta mechanism [DGLM95, LCD+94] as the basis of our language design, because Theta has important similarities to Java. Like Java, it uses declared rather than structural subtyping, and it also supports complete inter-module type checking. We rejected the C++ template mechanism [Sto87] and the Modula-3 [Nel91] generic module mechanism because they do not support our type-checking goal; they require that a generic module must be type checked separately for every distinct use. Furthermore, the most natural implementation of these mechanisms duplicates the code for different actual parameters, even when the code is almost entirely identical.

For the implementation, our concern was to achieve good performance for generic code in both execution speed and code size. An explicit goal was to use the same code for all instances of a generic abstraction, in order to avoid code blowup. Techniques for implementing parameterized code without bytecode extensions lead to slower execution or increased code size. Type checking our parameterized code in the Java bytecode verifier is also efficient: the code of a parametric class need only be verified once, like that of an ordinary class. All our extensions are backward compatible with the current JVM and have little impact on the performance of non-parameterized code. The result is that Java code is not only simpler to write, but also faster. For a simple collection class, avoiding the runtime casts from Object reduced run times by up to 17% in our prototype interpreter.

The remainder of the paper is organized as follows. Section 2 provides an informal description of our extensions to the Java language. Section 3 sketches how to implement the language extensions using the standard virtual machine. It also shows how these extensions are converted into our extended bytecode specification and how the extended bytecodes are verified and run. Finally, it presents some performance results. Section 4 discusses ways that other aspects of Java could be changed to take advantage of parametric polymorphism. We conclude in Section 5 with a discussion of what we have accomplished. The appendices contain detailed specifications of our language and virtual machine extensions.

2 Language Extensions

This section provides an informal description of our extensions to the Java language, illustrated with examples. It describes extensions to allow parameterized interfaces and implementations, and discusses several important design issues.

2.1 Java Overview

Java is similar to a safe version of C++ with all objects in the heap. Variables directly denote objects in the heap; there are no pointers. For safety, Java programs are statically type checked, and storage is managed by a garbage collector.

Java allows new types to be defined by both *interfaces* and classes [Sun95a]. An interface contains just the names and signatures of methods, but no implementations. A class defines methods and constructors, as well as fields and implementations.

An interface can *extend* one or more other interfaces, which means that the type defined by those interfaces are supertypes of the one being declared. Similarly, a class can extend another class (but just one), in which case it defines a subtype of the other class and also inherits its implementation.

Now we will discuss our extensions to support parameterized types.

```
interface SortedList [T]
    where T { boolean lt (T t); } ... { }
interface Map [Key, Value]
    where Key { boolean equals (Key k); } ... { }
```

Figure 1: Interface definitions

2.2 Parameterized Definitions

In our extended Java specification, interface and class definitions can be parameterized, allowing them to define a group of related types that have similar behavior but which differ in the types of objects they manipulate. All parameters are types; the definition indicates the number of parameters, and provides formal names for them. For example, the interface

```
interface SortedList [T] ... { }
```

might define sorted list types, which differ in the type of element stored in the list (e.g., SortedList[int] stores ints, while SortedList[String] stores strings). As a second example,

```
interface Map [Key, Value] ... { }
```

defines map types, such as Map[String,SortedList[int]], that map keys to values.

In general, a parameterized definition places certain requirements on its parameters. For example, a SortedList must be able to sort its elements; this means that the actual element type must provide an ordering on its elements. Similarly a Map must be able to compare keys to see if they are equal. The parameterized definition states such requirements explicitly by giving *where clauses*, which state the signatures of methods and constructors that objects of the actual parameter type must support. We discuss why we selected where clauses rather than other parameter constraint mechanisms in Section 2.8.

Figure 1 shows these two interfaces with their where clauses. In the definition of SortedList, the where clause indicates that a legal actual parameter must have a method named lt (less than) that takes a T argument and returns a boolean. The second definition states that a legal actual parameter for Key must have a method named equals that takes a Key as an argument and returns a boolean. Note that Map does not impose any constraints on the Value type, and therefore any type can be used for that parameter.

The body of a parameterized definition uses the type parameters to stand for the actual types that will be provided when the definitions are used. For example, Figure 2 gives the sorted list interface. Note the use of the parameter T to stand for the element type in the headers of the methods.

```
interface SortedList[T] where T {boolean lt (T t);} {
// overview: A SortedList is a mutable, ordered sequence.
//   Ordering is determined by the lt method of the element type.

    void insert (T x);
    // modifies: this
    // effects: adds x to this

    T first ( ) throws empty;
    // effects: if this is empty, throws empty,
    //   else returns the smallest element of this

    void rest ( ) throws empty;
    // modifies: this
    // effects: if this is empty, throws empty,
    //   else removes the smallest element of this

    boolean empty ( );
    // effects: if this is empty, returns true,
    //   else returns false.
}
```

Figure 2: The sorted list interface

Thus the insert method takes in an argument of type T, and the first method returns a result of type T.

2.3 Instantiation

A parameterized definition is used by providing actual types for each of its parameters; we call this an *instantiation*. The result of instantiation is a type; it has the constructors and methods listed in the definition, with signatures obtained from those in the definition by replacing occurrences of formal parameters with the corresponding actuals. We also use the term "instantiation" to refer to this type. Instantiations may be used wherever an ordinary type may be used.

When processing an instantiation, the compiler ensures that each actual parameter type *satisfies* the where clause for the associated parameter. This means that it has all the named methods and constructors, with compatible signatures. For example, the compiler rejects an instantiation such as SortedList[SortedList[int]], since the argument SortedList[int] does not have an lt method.

However, SortedList[int] *is* a valid instantiation, assuming that we can use existing primitive types such as int and char as actual type parameters even though these types do not have methods. We solve this problem in a simple and straightforward way: the various operations supported by these types are considered to be methods for the purpose of matching where clauses. For example, the == operator corresponds to the equals method, and < corresponds to the lt method. (See Section 4.1 for further discussion of primitive types.)

The information in the where clause serves to isolate the implementation of a parameterized definition from its uses (i.e., instantiations). Thus, the correctness of an instantiation of SortedList can be checked without having access to a class that implements SortedList; in fact, there could be many such classes. Within such a class, the only operations of the parameter type that may be used are the methods and constructors listed in the where clause for that parameter. Furthermore, it must use the routines in accordance with the signature information given in the where clause. The compiler enforces these restrictions when it compiles the class; the checking is done just once, no matter how many times the class is instantiated.

A legal instantiation sets up a binding between a method or constructor of the actual type, and the corresponding *where-routine*, the code actually called at runtime. Since an instantiation is legal only if it provides the needed routines, we can be sure they will be available to the code when it runs. Thus, the where clauses permit separate compilation of parameterized implementations and their uses, which is impossible in C++ and Modula 3.

2.4 Satisfying Where Clauses

An instantiation is legal if the actual parameter satisfies the where clause. Let us examine satisfaction more carefully. The compiler checks whether a type t (whether an interface, class, or formal parameter) satisfies a where clause for a method or constructor m by verifying that a hypothetical call to m would succeed. The arguments to this hypothetical call have the same types as in the where clause, but with all occurrences of the formal parameter replaced by t. For example, consider the where clause in the SortedList interface. The instantiation SortedList[X] is legal exactly when the statement b = x1.lt(x2) is legal, where b is a boolean and both x1 and x2 have type X.

Now, the call x1.lt(x2) is legal if the lt method of type X accepts any supertype of X, and returns any subtype of boolean (i.e., boolean itself). Therefore, an instantiation can be legal even if the signatures of constructors and methods of the actual type do not match exactly with those in the constraints. Instead we require only *compatibility*, using the standard contra/covariance rules for routine matching [Car88]. The only extension to these rules is that any exceptions thrown by the provided method must be subtypes of the exceptions given for that method in the where clause.

Because Java (like C++) allows overloading, sometimes a constraint is matched by more than one method or constructor of the actual type. In this case, the closest match is selected; if there is no uniquely best match, a compile-time error results. This is the same rule that is used in Java to decide which method/constructor to use in a call. For example, the unique method that would be called in the expression x1.lt(x2) is the one selected to satisfy the where clause.

Similar reasoning governs when protected and private methods can be used as where-routines: a where-routine binding is legal if the bound routine can be called legally at the point of instantiation. In our example, X would satisfy

```
interface SortedList_Member [T]
    where T { boolean lt (T); boolean equals (T); }
    extends SortedList[T] {

// overview: SortedList_Members are sorted lists
// with a membership test.

    boolean member (T x);
    // effects: returns true if x is in this else returns false.

}
```

Figure 3: Extending a parameterized interface

the where clause only if its lt method were visible to the code performing the instantiation.

So far we have considered only instantiations in which the actual parameter is a "known" type such as int or SortedList[int]. However, the actual in an instantiation can also be a formal type parameter. In this case, the satisfaction rule implies that the parameter satisfies a where clause only if it has a compatible where clause of its own. An example of such an instantiation will be given in the next section.

2.5 Extending Interfaces

In Java, interface and class definitions explicitly indicate their position in the type hierarchy. Parameterized definitions are no different. For example, the interface SortedList_Member given in Figure 3 extends SortedList by providing a membership test via the member method. Note that the where clause for this interface is more restrictive than SortedList's: the notion of membership requires an equality test (equals), and SortedList_Member objects also order their objects using lt.

The rule for the legality of an extension is particularly simple: An extension is legal if all the extended class or interface definitions are legal. The extended type must be a class or interface; it may not be a formal parameter. For example, the definition of SortedList_Member states that it extends SortedList[T]. Since the formal parameter T satisfies the lt where clause of SortedList, the declaration is legal.

The meaning of the extends clause is the following:

> For all types t where SortedList_Member [t] is a legal instantiation, SortedList_Member [t] extends SortedList [t]

Thus, SortedList_Member[int] extends SortedList[int], etc. If SortedList_Member[t] is legal, then we can be sure that SortedList[t] is also legal because of the type checking of the instantiation of SortedList when the SortedList_Member interface was compiled.

The extends clause need not use all the parameters of the interface being defined. Here are three examples:

```
interface A[T] extends B
interface Map[T, S] extends Set[T]
interface StringMap[T] extends Map[String, T]
```

In the first example, the declaration states that for any actual t, A[t] extends B. In other words, anywhere that a B is allowed, an A[t] can be used with any t (provided the instantiation is legal). In the second example, the declaration states that for any actual types t and s, Map[t,s] extends Set[t]. The final example shows that some parameters of the interface being extended may be supplied with actual types.

As in ordinary Java, all subtype relations for instantiations must be explicitly declared. The same rule holds for classes as well, except that Object is automatically a supertype of all class types. In particular, subtype relations between different instantiations of the same parameterized class are disallowed. Thus SortedList[BigNum] is *not* a subtype of SortedList[Num], even if BigNum is a subtype of Num. The reasons for this rule are discussed in Section 4.2 and elsewhere [DGLM95].

2.6 Implementations

A Java class can implement one or more interfaces, using the implements clause. The rule for legality of an implements declaration is the same as that for extends. For example, Figure 4 shows part of HashMap, a class that implements Map using a hash table. Note that the class requires equals and therefore the instantiation Map[Key,Value] is legal. The implementation of lookup uses both where-routines: hashcode is used to hash the input k to find the bucket for that key, and equals is used to find the entry for the key, if any. These calls are legal because the where clauses for Key describe them, and at runtime, the calls will invoke the corresponding operations of the actual parameter type being used.

A class need not implement an entire parameterized interface; instead, it can implement a particular instantiation, as shown in these examples:

```
class BigMap[Value] implements Map[long,Value] ...
class SortedList_char implements SortedList[char] ...
```

Such specialized implementations can be more efficient than general ones. For example, one fast implementation of SortedList_char uses a fixed-size array of integers, where each array element counts the number of occurrences of the corresponding character.

A parameterized class may have static variables such as the static variable count in HashMap. Each distinct instantiation of HashMap produces a distinct copy of this variable. For example, HashMap[String, Num] and HashMap[String, String] have separate count variables, but all instantiations HashMap[String,Num] share one static variable. In Java, static initializers of ordinary classes are run when the class is first used. Similarly, static initializers of an instantiation are run when the instantiation is first used.

2.7 Optional Methods

In addition to where clauses that apply to an entire interface or class, it is possible to attach where clauses directly to individ-

```
public class HashMap[Key,Value]
      where Key {
                boolean equals(Key k);
                int hashcode();
              }
      implements Map[Key,Value]  {

   HashBucket[Key,Value] buckets[ ]; // the hash array
   int sz; // size of the table
   static int count = 0; // number of hash tables

   public HashMap[Key,Value] (int sz) { . . . ; count++; }
   // effects:  makes this be a new,
   //          empty table of size sz.

   public Value lookup (Key k) throws not_in {
      HashBucket[Key, Value] b =
                 buckets[k.hashcode() % sz];
      while((b != null) && (!b.key.equals(k)))
        b = b.next;
      if (b == null)
        throw new not_in();
      else return b.value;
   }
   // other methods go here
}

class HashBucket[Key,Value] {
   public Key key;
   public Value value;
   public HashBucket[Key,Value] next;
}
```

Figure 4: Partial implementation of Map

ual methods, a feature originally provided by CLU [L+81]. Any where clause from the header of the class still applies, but additional methods can be required of the parameter type. A method that has additional where clauses is called an *optional method*, because it can be called only when the actual parameter has the method described by the where clause. If the parameter does not have the required method, the instantiation does not have the optional method. This condition can always be checked by the compiler; no extra runtime work is needed.

Optional methods are handy for constructing generic collection interfaces and classes. For example, a collection might have an output method that is present only if the elements have one too, as shown in Figure 5. The implementation of SortedList.output would use the output method of T to perform its job. The instantiation SortedList[Num] would be legal regardless of whether Num has an output method; however, it would have an output method only if Num had the method.

CLU also provided the ability to attach additional parameters to individual methods; while this functionality is useful in some cases, it adds substantial complexity to the design and implementation, so we chose to omit it.

```
interface SortedList[T]
        where T { boolean lt (T t); }
{
   . . .
   void output(OutputStream s)
      where T { void output (OutputStream s); }
      // effects: Send a textual representation of this to s.
}
```

Figure 5: An optional method

2.8 Alternate Constraint Mechanisms

Although we chose where clauses as the mechanism for constraining parameterized types, other choices are possible. We briefly examine some of these choices to show why they do not work well for extending Java. Further discussion of constraint mechanisms is available in an earlier paper [DGLM95].

2.8.1 Procedures

One simple approach is to avoid constraints and instead pass the needed routines explicitly when objects are created (as arguments to constructors — and the procedures would be stored in instance variables so that they would be accessible to the methods).

Even assuming that first-class procedures exist, passing procedures to constructors is not a good solution, since it doesn't properly reflect the semantics. For example, the meaning of a map depends very directly on the equality test used for Key. Two maps that use different notions of key equality are not really of the same type.

Finally, explicitly passing procedures prevents optimization (e.g., inlining) and uses space in each object to store the procedure.

2.8.2 Interfaces

Since where clauses resemble interfaces, it might seem that an interface could be used as the constraint; that is, the parameter would represent any type that is a subtype of the interface. The virtue of this approach is that some constraints can be expressed using the subtype notion that already exists; several languages have taken this approach [Mey88, MMPN93, S+86].

For example, we might try to write

```
interface SortedList[T] where T extends Ordered {
   . . .
}

interface Ordered {
   boolean lt(Ordered x);
}
```

The problem with this approach is that it cannot express constraints (such as lt) that mention the type parameter [DGLM95].

Because "lt" is a binary method, contravariance of arguments prevents most types from being subtypes of Ordered.

Note also that interfaces cannot express constructor and static method constraints, which are both possible with where clauses.

2.8.3 Parameterized Interfaces

To address some weaknesses of interfaces, parameterized interfaces can be used instead. This approach, related to F-bounded polymorphism [CCH+89], is used by some other statically-typed programming languages [KLMM94, OL92, Omo93]. In these schemes, the contravariance problem is avoided by adding parameters to the constraint interface. For example, the SortedList example would look as follows:

```
interface SortedList[T] where T extends Ordered[T] {
    . . .
}

interface Ordered[X] {
    boolean lt (X x);
    boolean equals (X x);
}
```

With this interface, any type T whose method signatures are compatible with Ordered[T] will also satisfy the constraint in its where clause form.

However, using parameterized interfaces as constraints does not work well in a language like Java, which has explicitly declared subtype relationships. For a type to be usable as a parameter argument to SortedList_Member, it must declare its relationship to Ordered and every other appropriate constraining interface. There will be many interfaces such as Ordered, each with its own combination of methods. In a large system, there may even be duplicate interfaces used as constraints. Every class will have to declare a multitude of interfaces that it satisfies.

Further, the need for a constraint may only be known later, because new parameterized classes (and constraints) can be added to the system. Existing types cannot be used as parameters because they do not declare their satisfaction of the new constraint. Sather [Omo93] addresses this problem by allowing new subtype declarations to be added to the system at any time. For example, after SortedList_Member is written, any users who wish to use SortedList_Member[Num] could add the following additional declaration:

```
class Num implements Ordered[Num];
```

This additional language extension seems likely to be problematic in a dynamically-loaded system like Java, where the full type hierarchy is not available.

If parameterized interfaces are used to constrain parameters, the subtype hierarchy is apt to become extremely complex as the system grows. The extra subtype relations do not serve any ordinary object-oriented purpose, since the constraint types are unlikely to be used other than as constraints. The number of subtype relations will be roughly proportional to the number of parameterized types and to the total number of types in the system. In most implementations of object-oriented systems, the resulting complex type hierarchy has a significant performance impact.

We might consider removing the requirement for an explicit subtype declaration. In this approach, the parameter matches the interfaces if its method signatures are compatible with the interface. In this case, a constraint interface no longer functions as an real interface, as there is probably no object providing the interface. Essentially, it becomes a clumsy way of bundling where clauses together.

A separate interface to constrain parameters might seem to offer economy of expression in some cases: one interface can be used to constrain several different parameterized definitions, rather than restating the where clauses for each definition. CLU [L+81] has a construct called a *typeset* that was provided for just this purpose. Yet, our experience with CLU is that typesets are almost never used — primarily because most constraints are very short (just one or two methods). Defining a separate interface just isn't worth the time.

3 Virtual Machine Extensions

In this section we discuss the implementation issues that arise in adding parameterized types to Java. We largely ignore the issue of compiling the extended Java language described in Section 2, since there are many languages with parametric polymorphism [MTH90, LCD+94, S+86]. Instead, we focus on what is new: extensions to the bytecodes of the Java Virtual Machine (JVM) that support parametric polymorphism, and the effect of these extensions on both the bytecode verifier and interpreter.

Extensions to the JVM are not required. Without JVM extensions, however, there are two options for implementation, both of which have some performance problems, as discussed in Section 3.1. For this reason, we have chosen to extend the virtual machine.

The Java compiler generates .class files, containing code in a bytecode format, along with other information needed to interpret and verify the code. The format of a .class file is described by the JVM specification [LY96]. We extended the JVM in a backwards-compatible way, as described in Section 3.2, so existing binaries can run without modification. The extended virtual machine supports not only the extended Java described in Section 2, but also could be used as a target architecture for other languages with subtyping or parametric polymorphism.

We show how to verify the extended bytecodes in Section 3.3 and how to interpret them in Section 3.4. Our implementation technique requires little duplication of information for each instantiation; in particular, bytecodes and global constants are not duplicated.

The extended bytecode interpreter has been implemented, showing that parametric polymorphism can be easily added to Java while preserving verification and speeding up some code. Code that does not use parametric polymorphism suffers little performance penalty. Performance results are given in Section 3.5.

3.1 Parameterization without JVM Extensions

There are two reasonable ways to implement parameterized types in Java without extending the virtual machine. We will briefly sketch these approaches, though we did not implement them because of their performance problems.

The most obvious implementation is to treat each instantiation of a parameterized class or interface as producing a separate class or interface. Each instantiation of a parameterized class has its own .class file that must be separately loaded into the interpreter and verified for correctness. In essence, the parameterized code is recompiled for each distinct set of parameters. This technique is similar to the template implementation used by most C++ compilers, which leads to substantial code blowup. It differs from the C++ approach in that the where clauses guarantee successful recompilation.

An alternative strategy produces code that is generic across all instantiations: the compiler can generate bytecodes for parameterized classes as though all parameters are of class Object. When compiling code that uses a parameterized class, the compiler generates runtime casts as appropriate. Because the compiler has type checked the code, all the runtime casts necessarily succeed, but the performance is the same as for old-style Java code that manipulates variables of type Object and performs explicit casts.

In this scheme, invocation of where-routines is complex. Each object of a parameterized class contains a pointer to a separate object that bundles up the appropriate where-routines for the instantiation, presenting them as methods. The compiler translates a where-routine invocation to an invocation of the corresponding method of this where-routine object. The where-routine object is installed in the object of a parameterized class by passing it as a hidden, extra argument to class constructors. The advantage of this technique over the previous one is that only the code of these where-routine objects is duplicated for each instantiation; most of the code of a parameterized class is shared for all instantiations.

This approach takes extra space in each object to point to the where-routine objects, and also adds the overhead of runtime casts. Gratuitous runtime casts occur at calls to methods of the parameterized class, and also within the methods of the where-routine objects. In addition, the already-noticeable cost of runtime casts can be expected to increase as more sophisticated interpreters and native code compilers are used. Native code compilation seems likely to improve simple operations like arithmetic and method invocation more than runtime casts, so the casts will become relatively more expensive. Therefore, in our implementation, we extended the bytecodes to express parameterization directly.

```
aload_2       // push b on stack
getfield      <Field HashBucket[Key,Value].key #0;>
aload_3       // push k on stack
invokewhere <Where Key.equals(#0;)Z> // call equals
```

Figure 6: Calling b.key.equals(k)

3.2 Bytecode Extensions

The most visible change to the virtual machine is the addition of two new opcodes (named invokewhere and invokestaticwhere) that support invocation of the methods that correspond to the where clauses. They provide invocation of normal and static methods, respectively. Constructors are treated as normal methods. For example, the expression b.key.equals(k) in Figure 4 is implemented using invokewhere, as shown in Figure 6.

Other minor changes were made to the format of a .class file, which supply the bytecode verifier with enough added information to directly verify parameterized code. We extended the encoding of type signatures to capture instantiation and formal parameter types, and added information to describe the parameters and where clauses of the class. Figure 6 shows one example of the extended type signatures: the signature for equals includes a "#0;", which represents the first formal parameter type (Key) of the current class. (The Z indicates that the return type of equals is boolean.)

3.3 Verifier Extensions

The JVM bytecodes are an instruction set that can be statically type checked. For example, bytecodes that invoke methods explicitly declare the expected argument and result types. The state in the execution model is stored in a set of local variables and on a stack. The types of each storage location can be determined by straightforward dataflow analysis that infers types for each stack entry and each local variable slot, at each point in the program.

It is important that extensions for parameterized types not remove the ability to statically type check bytecodes. The standard Java bytecode verifier works by verifying one class at a time [Yel95]. A call to a method of another class is checked using a declaration of the signature of that method, which is inserted in the .class file by the compiler. When the other class is loaded dynamically, these declarations are checked to ensure that they match the actual class signature.

Our extensions to the JVM preserve this efficient model of verification. The code of a parameterized or non-parameterized class is verified only once, in isolation from other classes, thus verifying it for all legal instantiations of the code. An instantiation of a class or interface can be checked for legality by examining only signature information in the

.class file for the parameterized class or interface; examining the bytecodes in the .class file is unnecessary.

Both the compiler and the verifier perform similar type checking, treating formal parameter types as ordinary types with a limited set of allowed operations. The important difference between the compiler and the verifier is that compiler variables have declared types, but the verifier must infer their types. The verifier must assign types to stack locations and local variable slots which are specific enough that an instruction can be type checked (e.g., if it invokes a method, the object must have that method). The assigned types must also be general enough that they include all possible results of the preceding instructions. For each instruction, the verifier records a type with this property, if possible. It uses standard iterative dataflow analysis techniques either to assign types to all stack locations for all instructions, or to detect that the program does not type check.

Because the bytecodes include branch instructions, different instructions may precede the execution of a particular instruction X. For type safety, the possible types of values placed on the stack by the preceding instructions must all be subtypes of the types expected by X. The core of the verifier performs this operation; it is a procedure to merge a set of types, producing the most specific supertype, or least upper bound in the type hierarchy. The dataflow analysis propagates this common supertype through X and sends its results on to the succeeding instruction(s). The analysis terminates when the types of all stack locations and local variable slots are stably assigned.

The primary change to the verifier for parameterized types is a modification to this merge procedure. To find the lowest common class in the hierarchy, we walk up the hierarchy from all the classes to be merged. Each time that a link is traversed to a superclass, we apply the parameter substitution that is described by the extends clause of the class definition. When a common class is reached in the hierarchy, the actual parameters of the common class must be equal.

Consider the class and interface hierarchy shown in Figure 7, with the corresponding extends clauses. The union of $B[X]$ and $C[X,Y]$ can be conservatively approximated by successively moving up the tree to find a common node while substituting parameters. The result is as follows:

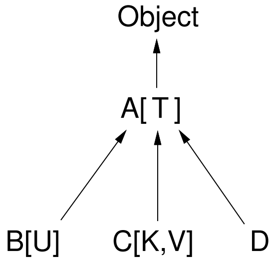
$$B[X] \cup C[X,Y] \subseteq A[X] \cup A[X] = A[X]$$

So these two types are merged to produce $A[X]$. Similarly, for $B[X]$ and $C[Y,X]$ we have

$$B[X] \cup C[Y,X] \subseteq A[X] \cup A[Y] \subseteq \text{Object}$$

In this case, the merge result is Object, since the parameters did not match for A. Note that unlike in the non-parameterized verifier, the lowest common superclass node is not always sufficient for the merge result, since it may be instantiated differently by the merged types. Finally, consider merging $B[B[\text{int}]]$ and D, which demonstrates that parameterized and non-parameterized classes can be merged:

$$B[B[\text{int}]] \cup D \subseteq A[B[\text{int}]]$$



$A[T]$ **extends** Object
$B[U]$ **extends** $A[U]$
$C[K,V]$ **extends** $A[K]$
D **extends** $A[B[\text{int}]]$
$E[T]$ **extends** Object

Figure 7: A parameterized class hierarchy

3.4 Runtime Extensions

The Java runtime implements the JVM, providing a bytecode interpreter and a dynamic loader for Java classes. We have produced a working prototype interpreter for our extensions to the JVM. Our design is based upon the Java runtime implementation provided by Sun Microsystems. Our goal in designing the runtime was to avoid duplication of information by the instantiations of a single class, while making the code run quickly. Minimizing the changes to the existing interpreter was also an important goal. Non-parameterized code runs about as fast as in the original interpreter.

3.4.1 Instantiation Pool

Classes are represented in the Java runtime by *class objects*. Each class points to its *constant pool*, which stores auxiliary constant values that are used by the Java bytecodes. These constant values include class names, field and method names and signatures. In this paper, constant pool entries are denoted by angle brackets. Figure 6 contains some examples of this notation, such as the field signature:

`<Field HashBucket[Key,Value].key #0;>`

For a parameterized class, some constants differ among the instantiations. For example, the code that is used by the where-clause operations differs among instantiations, and therefore cannot be placed in the constant pool. To resolve this problem, we create a new class object for each instantiation. Each *instantiation class object* stores instantiation-specific information in its *instantiation pool* (ipool). Values that are constant for all instantiations are still placed in the constant pool. Ipool indices are constant across all instantiations, but the contents of the corresponding slots differ. An instantiation object is created by cloning the parameterized class object and then installing a fresh ipool. The ipool serves a function similar to the where-routine object of Section 3.1.

For example, the HashMap method lookup from Figure 4 uses the equals operation of Key, so a pointer to the correct implementation of equals is placed in the ipool of each instantiation. Other examples of values in the ipool include references to classes that are instantiations that use the parameters, addresses of static (class) variables, and pointers to ipools of other instantiations.

This design allows us to duplicate very little data. All instantiations share the code of the class, the class method tables, and the constant pool. Only the data that is genuinely different for the instantiations is placed in the ipool, and there is exactly one class object for each instantiation being used in the running system.

3.4.2 Quick Instructions

The Sun implementation of the JVM includes an optimization that dynamically modifies the code the first time an instruction is executed, replacing some ordinary bytecodes with *quick* equivalents that do less work [LY96]. These quick instructions are not part of the bytecode specification and are not visible outside of the implementation. The non-quick version of such an instruction performs a one-time initialization, including dynamic type checking, and then turns itself into the quick version. Our implementation makes extensive use of this self-modifying code technique in order to improve performance. A more detailed discussion of the implementation technique is available [BLM96].

The presence of subclasses makes ipools substantially more complicated than implementing parametric polymorphism for languages like CLU or ML [MTH90], which are not object-oriented. Consider a method call: the compiler and dynamic linker cannot tell which piece of code is run, so they cannot determine the corresponding ipool to use, either. The proper ipool to use for the call is not known until the actual call, and must be determined from the object on which the method is being invoked (the method receiver). The object has a pointer to its instantiation class object, which contains the ipool for the methods of that class. Since the method may be inherited from another class, an additional indirection is required to obtain the correct ipool.

Our implementation technique resembles an earlier approach [DGLM95], but extended to provide support for static methods, static variables, and dynamic type discrimination. It requires few changes to the existing Java interpreter, and is also applicable to compiled machine code[1].

[1] Suitably extended, the earlier approach probably produces faster machine code.

3.4.3 Primitive Types

Primitive types such as integers can be used as parameter types under the implementation technique described here. Most of the primitive types take up the same space in an object as an object reference. That they are not full-fledged objects is not a problem for invoking where-routines, since where-routines are accessed through the ipool rather than through the object.

Variables whose type is a parameter are treated in the bytecodes as though they are object references; for example, the code of Figure 6 will work properly even when the code is instantiated on Key = int, although an aload instruction is used in this case to access an integer. Other bytecodes such as putfield and getfield do not distinguish between references and primitive values, and do not create a problem.

The same code cannot support instantiation on long and double, since these data values do not occupy the same space as an object reference. The best way to handle large primitive types such as long and double is probably to rewrite instantiated code when those types are used as actual parameters. Note that the specialized instantiations might not even have the same memory layout, since instance variables of type long will take up more space than other parameter types. The only problem with this technique is that the specialized instantiations for double and long would not share code with the other instantiations of the same class, which would all use the same bytecodes.

3.5 Performance Results

We performed a few simple benchmarks to investigate the performance of our extended interpreter. These results must be considered preliminary, since we have not tuned our interpreter performance. Also, since the Java interpreter we were modifying runs at only a fraction of the speed of machine code, these results cannot be definitive. However, the results suggest that parameterized code adds little overhead to Java and offers performance improvements when its use is appropriate.

The results of our benchmarks are shown in Figures 8, 10, and 11. The benchmarks were run several times, so measurement variation is smaller than the significant digits shown. All run times are in seconds.

First, we confirmed that our extended interpreter did not significantly slow the execution of ordinary unparameterized code, by running the Java compiler javac (itself a large Java program) on both interpreters. As shown in Figure 8, the extended interpreter runs only 2% slower than the original interpreter on non-parameterized code.

We also ran some small benchmarks to compare the speed of parameterized and non-parameterized code, using a simple parameterized collection class and some non-parameterized equivalents. In these comparisons, the code was almost iden-

Original interpreter:	8.5
Extended interpreter:	8.7

Figure 8: Java Compiler Speed (in seconds)

```
Cell[Element] c = new Cell[Element]();
c.add(new Element());
for (i = 0; i < 1000000; i++) {
  Element t = c.get();
  t.do_method();
}
```

Figure 9: Micro-benchmark code

tical, except for changes in type declarations and a dynamic cast. The code used repeatedly extracts an object from a parameterized collection and invokes a method on that object, as shown in Figure 9.

In the first micro-benchmark, we compared the performance of the parameterized class Cell[T] to a hard-wired ElementCell class, which can only contain objects of type Element. Both of these classes represent a simple collection that contains only one element. Using this simple collection isolates the performance effects that we want to measure. The hard-wired class, which is less generally useful, will obviously run faster. But as Figure 10 shows, there is only a 4% penalty for running parameterized code.

In the second micro-benchmark, we compared the parameterized class to the same collection class where the types of the components are all Object — the option for old-style Java code that the existing Java utility classes mostly follow. This version of the collection class gives up some static type checking, and also requires that the programmer write an explicit type cast when extracting values from the collection (in order to call do_method). Figure 10 shows that for our benchmark, this approach is 17% slower than using parameterized types.

In the third micro-benchmark, we compared the speed of invoking where-routines (with the invokewhere bytecode), ordinary methods (with invokevirtual), and interface methods (with invokeinterface). Our test program intensively called these respective bytecodes. The results are shown in Figure 11. In our results, all three method invocation paths are within 6% of each other. This experiment represented a best case for invokeinterface, since its inline caching worked perfectly.

Although these performance results are produced by modifying an interpreter, we believe that the advantages of the

Parameterized:	13.3
Hard-Wired Types:	12.8
Using Object:	15.5

Figure 10: Collection Class Results

invokewhere	22.8
invokevirtual	21.9
invokeinterface	23.3

Figure 11: Invocation Speed

extended bytecodes and of parameterized code will only increase with compiled machine code, since the interpreter tends to add a constant overhead to all operations. The slight overhead of Figure 8 is not intrinsic to parameterized code and can be eliminated with more careful implementation. The advantage of parameterized code shown in Figure 10 will be more dramatic, particularly since the cost of the runtime cast from Object will be relatively more expensive than method invocation.

4 Interaction of Parameterization with Java

This section discusses some ways in which the addition of parameterization to Java interacts with Java features.

4.1 Primitive Types

In order for primitive types such as int to satisfy where clauses, it is necessary for them to have methods. Therefore, we have assumed that they do indeed have methods corresponding to the usual infix notation, at least for the purpose of matches where clauses.

In addition, it is important to use common naming conventions for the methods of the primitive types. The reason for this requirement is that matching in where clauses is based on method names, and standard naming conventions make more matches possible.

4.2 Java Arrays

While the current Java language does not support parameterization of classes or interfaces, there is support for parameterized arrays. Unfortunately, the subtyping rule for Java arrays is different from the subtyping rule we propose in Section 3 for parameterized classes and interfaces.

The subtyping rule for Java arrays *does* allow subtypes that are covariant in the parameter types. If S is a subtype of T, an array of S (written S[ ]) is a subtype of an array of T (T[ ]). This rule has the consequence that array stores require a runtime type check, and this check is in fact made by the Java runtime. For example, consider the following code:

```
     T x = ...;
*    T [ ] z = new S [ ] (...);
     ...;
**   z[i] = x;
```

The assignment (*) is legal because S is a subtype of T and therefore S [ ] is a subtype of T [ ]. The assignment (**)

is also legal (at compile time) because it is legal to store a T object into a T[]. However, if this store were allowed to happen, the result would be that an S[] contains an element whose type is not a subtype of S. Therefore the store cannot be permitted, and Java prevents it by checking at runtime that the actual type of the object being stored is a subtype of the element type of the array[2].

The motivation for the Java subtyping rule seems to be that it is better to check stores at runtime than to copy the entire array when a conversion from S[] to T[] is desired. While this may be true for some programs, it is not difficult to imagine programs in which stores are a dominant cost; for general parameterized types, it is even easier to imagine costly situations, since every method that takes an argument of a parameter type would require the check.

We believe that for general parameterization of classes and interfaces it is better not to allow covariant-parameter subtyping. It would be possible to allow such subtyping if no methods of the supertype take arguments of the parameter type, but such types are rare. (A more complete discussion of these issues is available [DGLM95].) Furthermore, covariant-parameter subtyping creates implementation difficulties for efficient dispatch mechanisms [Str87, Mye95]. We believe that the Java rule for arrays should be changed so that all parameterized types have the same rule. However, our design does not require this; different rules can be used for arrays and for other parameterized types.

4.3 The Java Type Library

Java is being augmented with a library of interfaces and classes. Already there are several abstractions in the library that would be more useful if parameterization were available. Types such as Vector and Enumeration are most naturally written as parameterized types, but currently manipulate objects of type Object instead.

This has two unfortunate consequences:

1. When new elements are added to a vector, or are stored in the vector, if they are of a primitive type such as int then it is necessary to to *objectify* them. For example, int must be explicitly wrapped in an Integer object to make it usable as an Object. This wrapping step is awkward for the programmer and has runtime overhead.

2. Whenever an element is fetched from a Vector or produced by an Enumeration, it must be explicitly cast from Object to the expected type. If the element has a primitive type, it also must be unwrapped after the cast, adding even more cost and coding complexity.

[2]The check can be avoided if the compiler can figure out what is going on at compile time, or if T has no subtypes. Both primitive types and final classes satisfy this latter property, but Sun's current Java interpreter only removes the type check for the primitive type case.

Both problems are ameliorated by parameterized types. When elements are added, there is no need to objectify them, and when they are retrieved from the collection, there is no need for the expensive runtime cast or unwrapping.

5 Conclusions

Java offers the real possibility that most programs can be written in a type-safe language. However, for Java to be broadly useful, it needs to have more expressive power than it does at present.

This paper addresses one of the areas where more power is needed. It extends Java with a mechanism for parametric polymorphism, which allows the definition and implementation of generic abstractions. The paper gives a complete design for the extended language. The proposed extension is small and conservative and the paper discusses the rationale for many of our decisions. The extension does have some impact on other parts of Java, especially Java arrays, and the Java class library.

The paper also explains how to implement the extensions. We first sketched two designs that do not change the JVM, but sacrifice some space or time performance. Our implementation avoids these performance problems. We had three main goals: to allow all instantiations to share the same bytecodes (avoiding code blowup), to have good performance when using parameterized code, and to have little impact on the performance of code that does not use parameterization.

The implementation discussed in Section 3 meets these goals. In that section, we described some small extensions to the virtual machine specification that are needed to support parameterized abstractions; we also described the designs of the bytecode verifier and interpreter, and the runtime structures they rely on.

Preliminary performance results from our implementation of the extended bytecode interpreter show roughly a 2% penalty for the presence of parameterized code, but a speedup for parameterized code of 17%, by eliminating runtime checks. We expect that some simple performance tuning can improve these results.

The appendices that follow present a more detailed specification of our extensions to the Java language and to the Java virtual machine.

Parameterized abstractions are not the only extension needed to make Java into a convenient general purpose programming language. First-class procedures and iterators would also be valuable extensions to the language. However, we believe that parameterized types are the most important feature currently missing from Java.

Acknowledgements

We would like to thank Kavita Bala, Bill Joy, Matt Kennel, and the reviewers for their helpful comments.

A Java Grammar Extensions

This appendix presents the full syntax for our extension to Java, and discusses some issues of the semantics that were not discussed earlier. Appendix B provides the details of the extensions to the Java Virtual Machine that support parametric polymorphism.

In the syntax below, we use the brackets [and] when zero or one occurrence of the construct is allowed (i.e., when the construct is optional). We have not attempted to give a full syntax for Java, but rather we just focus on the parts that are affected by the addition of parameterization. Capitalized non-terminals are defined by the Java language grammar [Sun95a] and are not repeated here.

A.1 Interfaces

An interface definition can be parameterized by using the following syntax:

```
interface        →  [InterfaceModifiers] interface idn
                    [ [ params ] ] [where]
                    [ExtendsInterfaces] InterfaceBody

params           →  idn [, idn]*

where            →  where constraint [, constraint]*

constraint       →  idn [<whereDecl> ;]*

whereDecl        →  methodSig | constructorSig

methodSig        →  [ static ] ResultType
                    MethodDeclarator [Throws]

constructorSig   →  ConstructorDeclarator [Throws]
```

The *params* clause lists one or more formal parameter names, each of which stands for a type in the parameterized definition. The *where* clause lists constraints on the parameters. Each constraint identifies the parameter being constrained; the *idn* in the constraint must be one of those listed in the *params*. It then identifies the methods/constructors that that parameter must have. For a method it gives the name, and the types of its arguments and results; it also indicates whether the method is static or not. For a constructor, it lists the types of the arguments. The type names introduced in the *params* clause can be used as types in the remainder of the interface, including the *ExtendsInterfaces* clause.

A.2 Instantiation

The form of an instantiation is

```
instantiationType  →  idn [ actualParams ]
```

where

```
actualParams  →  Type [, Type ]*
```

The *idn* in the instantiation must name a parameterized class or interface. The number of *actualParams* must match the number given in that parameterized definition, and each *Type* must satisfy the constraints for the corresponding formal parameter of that definition.

Satisfying a constraint means: (1) if the constraint indicates a constructor is required, the actual type must have a constructor with a signature that is compatible with that given in the constraint; (2) if the constraint indicates that a method is required, the actual type must have a method of that name with a signature that is compatible with that given in the constraint; if the constraint indicates the method is static, the matching method in the actual type must also be static, and otherwise it must not be static.

A.3 Classes

A class definition can be parameterized by using the following syntax:

```
ClassDeclaration  →  [ClassModifiers] class idn
                     [ [ params ] ] [where] [Super]
                     [Interfaces] ClassBody
```

As was the case for interfaces, the *params* of the class can be used as types within the class. Also, code in the *ClassBody* can call the routines introduced in the *where* clause. Such a call is legal provided the routine being called is introduced in the where clause and has a signature that matches the use. The syntax of the call depends on what kind of routine is being called: for an ordinary method, the syntax `x.m( )` is used to call the method; for a constructor, the syntax `new T( )` is used; and for a static method the syntax `T.m( )` is used.

A.4 Methods

Methods can have where clauses of their own:

```
MethodDeclaration  →  [MethodModifiers] ResultType
                      MethodDeclarator [Throws]
                      [where] MethodBody
```

The *where* clause can constrain one of the type parameters of the containing interface or type. Calls to methods or constructors of formal parameter types are legal within the method body if they match constraints for the parameter that are given either in the *where* clauses of the containing class or interface, or in the *where* clauses of the method.

B JVM Extensions

This appendix describes the extensions to the Java Virtual Machine that support parametric polymorphism. In the following specification, we include section numbers from the original JVM specification [Sun95b] to indicate where changes are being made.

(1.4) Objects whose type is statically understood to be a parameter type are always stored as 32-bit quantities on the stack and in local variables. The aload and astore bytecodes are used to access these variables, effectively treating them as references. The types double and long cannot be used as type parameters at the VM level. The Java compiler will either automatically wrap these data into Double and Long objects respectively, or automatically instantiate such classes, rewriting the code appropriately. The only exception to this rule is arrays, where longs and doubles are stored in packed form.

(2.1) The class file format is extended to contain the following additional field:

```
parameter_info parameters;
```

where parameter_info is defined as follows:

```
parameter_info {
        u2 parameters_count;
        u2 parameter_names[parameters_count];

        u2 where_count;
        u2 where_clauses[where_count];
}
```

The array parameter_names contains constant pool indices for the names of the formal parameters. These name are provided for debugging and disassembly purposes, as in Figure 6.

The field where_clauses contains constant pool indices for entries of type CONSTANT_WhereRef, which describe one where clause of a indicated parameter as specified below. Its signature may mention the parameter types.

(2.2) Signature specifications include the following additional options:

```
M<fullclassname>[<fieldtype>...]
```

This signature denotes the instantiation of the parameterized class or interface using the types inside the brackets. The number of parameters much match the number of parameters in the class, and the parameter types must have the required methods.

```
#<parameterindex>;
```

This signature denotes one of the parameter types of the current class or method. The parameter index is written as an integer in ASCII form.

Note that these new grammar productions inductively preserve the property that the end of a typespec can be determined as you read the characters from left to right.

(2.3.1) For example, the string "MMutex[[I]" represents the type Mutex[int[]], and inside a class Set[Object], the string "[#0;" represents an array of Object and "MHashMap[#0;I]" represents a HashMap[Object, int].

(2.3.2) The new constant pool entries CONSTANT_LargeMethodref and CONSTANT_LargeFieldRef are similar to CONSTANT_Methodref and CONSTANT_FieldRef except that that they take up two constant pool entries. Their format in the class file is exactly the same as CONSTANT_Methodref. They are used to refer to static methods and fields of parameterized classes.

A LargeMethodref must be used if a method is used by a static or non-virtual method call, and the actual implementation of the method is defined in a parameterized class. This condition can be checked by the compiler. For example, if the class signature described by constant_pool[class_index] is an entry of type CONSTANT_Class that describes an instantiation, and the method is static, then a LargeMethodref must be used. However, a LargeMethodref is required even when constant_pool[class_index] is not an instantiation, but it inherits its implementation of the method from a parameterized class. The extra constant-pool entry is used to find the correct ipool for the static method code.

A LargeFieldref must be used for all accesses to a static field of a parameterized class. The extra constant-pool entry contains an ipool index; at that index, the ipool contains a pointer to the static variable storage. This mechanism allows each distinct instantiation of a parameterized class to have its own copy of the static variable.

(2.3.8) CONSTANT_WhereRef is a new kind of constant pool entry, used to denote an operation of a parameter type. It corresponds to a where clause.

```
CONSTANT_WhereRef {
    u1 tag;
    u2 param_index;
    u2 name_and_type_index;
    u2 access_flags;
}
```

The tag will have the value CONSTANT_WhereRef. The param_index is the index of the parameter type in the class or the method that calls this parameter operation. The name_and_type_index describes the signature and name of the operation, as described in the where clause. It must match the signature in parameters[param_index].where_clauses[k].methods, above.

The access_flags field may only have ACC_STATIC set of the various possible flags.

(2.5) The structure method_info, which describes one method of the current class, is extended to contain the following field:

```
parameter_info parameters;
```

which describes any new parameters that apply within the scope of the method and where clauses on both these parameters and on any existing parameters. The parameters and where clauses of the class also apply to the method. New parameters are indexed sequentially following the indices of the class parameters, so different methods that have their own additional type parameters may use the same indices. However, there is no ambiguity in any given scope about which parameter corresponds to an index. The current prototype does not implement additional method parameters.

(3.15) Two new bytecodes are added to the virtual machine for invoking where-clause operations. New bytecodes are not strictly necessary, since the attributes of the constant pool entry serve to disambiguate the various invocation bytecodes. We added new bytecodes by analogy with existing bytecodes, and our interpreter does have distinct quick bytecodes.

invokewhere

Invoke an operation of a parameter type.

Syntax:

invokewhere = 186
indexbyte1
indexbyte2

Stack: *..., objectref, [arg1, [arg2 ...]]* ⇒ *...*

The index bytes are used to form an index into the constant pool, which must be an entry of type CONSTANT_WhereRef. This entry is used to determine which code to run for the method, using information in the current object to determine what the parameter type is.

invokestaticwhere

Invoke a static operation of a parameter type.

Syntax:

invokestaticwhere = 187
indexbyte1
indexbyte2

Stack: *..., objectref, [arg1, [arg2 ...]]* ⇒ *...*

The index bytes are used to form an index into the constant pool, which must be an entry of type CONSTANT_WhereRef. This entry is used to determine which code to run for the method, using information in the current object to determine what the parameter type is.

References

[BLM96] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. Technical Memo MIT/LCS/TM-553, Massachusetts Institute of Technology, 1996.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[CCH+89] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. of OOPSLA '95*, pages 156–158, Austin TX, October 1995. ACM SIGPLAN Notices 30(10).

[KLMM94] Dinesh Katiyar, David Luckham, John Mitchell, and Sigurd Melda. Polymorphism and subtyping in interfaces. *ACM SIGPLAN Notices*, 29(9):22–34, August 1994.

[L+81] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[MMPN93] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mye95] Andrew C. Myers. Bidirectional object layout for separate compilation. In *Proc. of OOPSLA '95*, pages 124–139, October 1995. ACM SIGPLAN Notices 30(10).

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[OL92] Stephen M. Omohundro and Chu-Cheow Lim. The Sather language and libraries. Technical Report TR-92-017, International Computer Science Institute, Berkeley, March 1992.

[Omo93] Stephen M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, October 1993.

[S+86] C. Schaffert et al. An introduction to Trellis/Owl. In *Proc. of OOPSLA '86*, Portland, OR, September 1986.

[Sto87] B. Stoustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[Str87] B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 European Unix Systems User's Group Conference*, Helsinki, Finland, May 1987.

[Sun95a] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at ftp://ftp.javasoft.com/docs/javaspec.ps.zip.

[Sun95b] Sun Microsystems. *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995. Available at ftp://ftp.javasoft.com/docs/vmspec.ps.zip.

[Yel95] Frank Yellin. Low-level security in Java, December 1995. Presented at the Fourth International World Wide Web Conference, Dec. 1995.