MIT LCS TM-553

Parameterized Types and Java

Joseph A. Bank Barbara Liskov Andrew C. Myers

(alphabetically)

Programming Methodology Group

May 1996

Parameterized Types and Java

Joseph A. Bank Barbara Liskov Andrew C. Myers

(alphabetically)

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139
{jbank,liskov,andru}@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. The paper discusses both the rationale for our design decisions and the impact of the extension on other parts of Java, including arrays and the class library. It also describes an implementation of the mechanism, including extensions to the Java virtual machine, and designs for the bytecode verifier and interpreter. The bytecode interpreter has been implemented; it provides good performance for parameterized code in both execution speed and code size, and does not slow down programs that do not use parameterized code.

1 Introduction

Java [Sun95a] is an interesting programming language because of its potential for WWW applications. It is additionally interesting because it is a type-safe object-oriented language. We believe that it would be extremely helpful to our profession if most programming were done in such a language. For the first time, we have a chance of having this happen. Because of the widespread interest in Java, its similarity to C and C++, and its industrial support, many organizations may switch to Java from their current language of choice for implementing most applications.

Java is also interesting because of its heterogenous target architecture, the Java Virtual Machine (JVM) [Sun95b]. The virtual machine executes typed bytecodes that can be checked by a bytecode verifier before they are run. Verification of compiled Java code is a key reason why Java is attractive for web applications. Because the JVM bytecodes can be statically type checked, Java programs from untrusted sources can be used without fear that they will violate type security to access private information or disrupt the execution of the interpreter. Typed bytecodes may also help in compiling to efficient machine code.

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 the form of parametric polymorphism. 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 parametric abstraction by using runtime type discrimination. For example, the standard hash table interface maps keys of type Object to values of type Object. Any values returned from a hash table must be explicitly cast down to the expected type, which is onerous for the programmer, and requires relatively expensive work at runtime to ensure that the cast is safe.

This paper extends Java with parametric polymorphism, a mechanism for writing generic interfaces and implementations. It provides a complete design of this extension and shows how to efficiently implement this design by extending the Java Virtual Machine bytecodes, the bytecode verifier and the interpreter. These extensions are not strictly necessary but lead to better performance. The paper presents results from a working prototype of the extended JVM interpreter that show

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.

our technique adds little space or time overhead. The paper also discusses the interaction of parametric polymorphism with other Java features and identifies a few useful modifications to Java.

An explicit goal of our work was to be very conservative. For the language extensions we took a mechanism that works and adapted it to Java with as few changes as possible. Our approach was guided by the desire to support the Java philosphy of providing separate compilation with complete inter-module type checking; this seemed important both for pragmatic reasons, and because it is consistent with the approach in Java.

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. In addition, our changes to the Java bytecode interpreter are simple and resulted in little additional overhead. Typechecking 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 upward compatible with the current JVM and have little impact on the performance of non-parameterized code. Finally, we achieve performance improvements along with simpler Java code. For a simple collection class, avoiding the runtime casts from Object reduced run times by up to 16% in our prototype interpreter.

We used the Theta mechanism [DGLM95, LCD$^+$94] as the basis of our language design. Theta was a good starting point because it uses declared rather than structural subtyping, as does Java, and because it supports complete inter-module type checking. We do not discuss in detail why this mechanism was chosen because this discussion has been presented elsewhere [DGLM95]. We rejected the C++ template mechanism [ES90] and Modula-3 [Nel91] generic module mechanism because they do not support our type-checking goal. These mechanisms do not support separate compilation because a generic module must be type-checked separately for every distinct use. Furthermore, the most natural implementation of these mechanisms (and the one that is actually in use in these systems) treats the code as a kind of macro, so that code is typically duplicated for different actual parameters, even when the code is almost entirely identical.

The remainder of the paper is organized as follows. Section 2 provides an informal description of our extensions to the Java language. Section 3 shows how these extensions are converted into our extended bytecode specification and how the extended bytecodes are verified and run. It also 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.

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. Appendix A gives the syntax of the extension.

2.1 Java Overview

We do not assume any great familiarity with Java on the part of the reader, and therefore we need to say a few words about Java before we start. Java is similar to C++ with the unsafe features eliminated. The most fundamental difference is that most Java objects are in the heap, which is managed automatically by a garbage collector, and variables directly denote heap objects.

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

An interface can *extend* one or more other interfaces, which means the types defined by those interfaces are supertypes of the one being declared. Similarly, a class can extend another class (but just one), meaning that it defines a subtype of the other class *and* also inherits the implementation. Thus, Java allows multiple supertypes, but only single inheritance. In addition, a class can implement one or more interfaces; this means that (in conjunction with the code inherited from its superclass, if any) it implements all the methods of the interfaces. Java supports the usual rules for assignment. For example, the code

```
C x = ...;
D y = ...;
x = y;
```

is legal provided D is a subtype of C (there are some oddities here but they do not affect the discussion in this paper).

Now we will move on to discuss our extensions to Java that support parameterized types.

2.2 Parameterized Definitions

Interface and class definitions can be parameterized, allowing them to define a group of related types that have similar behavior but that differ in the types of elements they contain. All parameters are types; the definition indicates the number of parameters, and provides formal names for them.[1] 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), but which all provide sorted access to the elements. As a second example,

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

defines map types, such as Map[String,SortedList[int]], that map from 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 identify methods and constructors that the actual parameter type must have, and also state their signatures. We will not discuss other mechanisms for constraining parameters, because the merits of where clauses are covered elsewhere [DGLM95].

Here are the above definitions with their where clauses:

[1]It is possible to allow other kinds of parameters than types (for example this is done in CLU [LSAS77]), but we have found them to not be very important; in fact they just duplicate the ordinary parameter passing mechanism. If other kinds of parameters are allowed, their objects must be immutable and the actual parameter values must be known at compile time; thus arrays could not be parameters because they are mutable.

```
interface SortedList[T] where T {boolean lt (T t);} {
// overview: A SortedList is a mutable, ordered set where
//   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 1: The sorted list interface

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

In the first definition, the where clause indicates that a legal actual parameter must have a method named lt 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 1 gives the sorted list interface. Note the use of the parameter T to stand for the element type in the headers of the methods. 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 *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. For example, the instantiation SortedList[int] has the following methods:

```
void insert (int);
int first ( ) throws empty;
void rest ( ) throws empty;
boolean empty ( );
```

When processing an instantiation, the compiler makes sure that each actual parameter type *satisfies* the where clause for the associated parameter. This means that it has all the constructors listed with the required signatures, and it has methods with the given names and signatures (actually, the signatures need to be *compatible*, as discussed below.) Furthermore, if the where clause requires a static method, the actual must have a corresponding static method. In the case of built-in types like int and char, Java does not define any methods; instead, these objects have infix operators, e.g., == and +. Therefore, we assume an extension to Java: built-in types have the obvious methods, which can be used for matching in instantiations, even if the language does not allow them to be used in calls. Thus int has methods called lt and equals, and these correspond to the code that runs when the related infix notation is used.

On the other hand, an instantiation is not legal, and is rejected by the compiler, if the actual parameter does not satisfy the constraints given in the where clause. For example, SortedList[SortedList[int]] is not a legal instantiation because the argument, SortedList[int], does not have an lt method.

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, and in fact there could be many such classes. Within such a class, code is allowed to use the methods/constructors listed in the where clause for that parameter, and *only* those routines. 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.

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 provide separate compilation of implementations and uses of parameterized definitions. (Furthermore, it is the lack of something like where clauses that makes separate compilation with complete inter-module type checking impossible in C++ and Modula 3.)

An instantiation can be legal even if the signatures of constructors and methods of the actual type do not match exactly with those listed in the constraints. Instead we require only *compatibility*. The intuition here is that when code implementing the parameterized definition calls methods/constructors listed in the where clause, the call will actually go to a method/constructor provided as part of the instantiation, and that call must be type correct at runtime. Therefore, the argument types of the method/constructor provided by the instantiation can be supertypes of what is stated in the where clause, the result type of a provided method can be a subtype of what is stated in the where clause, and any exceptions thrown by the provided method must be subtypes of the exceptions given for that method in the where clause. (These are the standard contra/covariance rules for routine matching [Car84] except that we have extended them to work for Java exceptions.) For example, suppose BigNum is a class with method

```
boolean lt (Num);
```

where Num is a superclass of BigNum. Then SortedList[BigNum] is a legal instantiation because the signature of BigNum's lt method is compatible with boolean lt (BigNum t), which is the where clause of SortedList with BigNum substituted for T.

A legal instantiation must set up a binding between a method/constructor of the actual type, and the corresponding *where-routine*, the code actually called at runtime; we will discuss how this is done in Section 3. Because Java (like C++) allows overloading, there can sometimes be more than one method/constructor of the actual type that matches a particular constraint. 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. (It matters which method/constructor is selected because this is the one that will actually be called when the implementation runs.) For example, suppose BigNum has two lt methods:

```
boolean lt (BigNum);
boolean lt (Num);
```

and consider the call x.lt(y) where x and y are declared to be of type BigNum. Java will select the first definition, and we will likewise select the first definition for the instantiation SortedList[BigNum].

The same reasoning governs when protected and private methods be used as where-routines: if a call can be made in some environment using arguments of the types specified by the where clause, then a corresponding instantiation is legal, and the method used for the call is bound to the where clause. For example, a where clause can be satisfied by a protected method if the instantiation takes place in the code of a subclass, where the protected method is accessible.

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 where clause of that parameter is used to determine whether the instantiation is legal, and the where-routine of that parameter is then bound to the corresponding where-routine for the formal. An example of such an instantiation will be given in the next section.

Finally, we need to state the rules for determining type equality for types produced by instantiation. These rules are straightforward: Two instantiations define the same type iff they instantiate the same parameterized definition, and their arguments are pairwise equal.

2.4 Extending Interfaces

A parameterized definition can indicate its place in the hierarchy (just as is done in a non-parameterized definition). For example, the interface given in Figure 2 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, and in addition SortedList_Member objects contain their elements in sorted order.

The header of SortedList_Member states that it extends SortedList[T]. First, note that this is a legal instantiation: T satisfies the constraint for SortedList since SortedList_Member requires an lt method for

```
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 2: Extending a parameterized interface

it. Second, the meaning of the extends clause is the following:

> For all types t where SortedList [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 SortedList[T] when the SortedList_Member interface was compiled.

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

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

In the first case, 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 case, the declaration states that for any actual types t and s, Map[t,s] extends Set[t]. Probably such definitions do not occur very often, but they are useful occasionally. For example, we might want to define a hierarchy of interfaces with a top element, top, a type with some standard methods that many objects share. Then we will be able to say that all instantiations of a parameterized interface extend top.

All subtype relations for the types obtained from parameterized interfaces must be explicitly declared. This same rule holds for classes as well, except that Object is automatically a supertype of all class types. Furthermore, we do not allow for subtype relations between different instantiations of the same parameterized class. covariant-parameter subtypes. Thus SortedList[BigNum] is *not* a subtype of SortedList[Num], even if BigNum is a subtype of Num. We discuss this avoidance of *covariant-parameter subtyping* in Section 4, when we compare our rule with the rule for Java arrays.

2.5 Implementations

A Java class can implement one or more interfaces. For example, Figure 3 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 of the information in the where clause for Key, and they will go to the where-routines provided for the particular instantiation being used when the lookup method is called (as discussed in Section 3).

Objects are obtained in Java by calling constructors. For example, in our extension

```
Map[String,Num] m =
  new HashMap[String,Num](1000);
```

creates a new HashMap object and assigns it to m, a variable of type Map[String,Num]; the assignment is legal because HashMap[String,Num] implements Map[String,Num]. The main point to note here is that it is clear what instantiation an object belongs to when it is created; this is important because the implementation relies on it to find the where-routines when methods are called on the object. Another point is that the use of the HashMap constructor is legal because of our assumption that built-in types all have certain common methods; we assume hashcode is such a method since all Java objects inherit it from Object.

A parameterized class might have static variables, e.g., HashMap might have a static variable count that keeps track of the number of HashMaps that have been created. Each distinct instantiation of HashMap would have its own copy of this variable. For example, HashMap[String,Num] and HashMap[String,String] would have separate count variables, but all instantiations HashMap[String,Num] would share one static variable. Like static initializers of ordinary classes, which are run when the class is first used, static initializers of an instantiation are run when the instantiation is first used. In some cases, one might want a static variable that is shared by all instantiations of a class. This functionality can be achieved by placing the static variable in a separate unparameterized class in the same module.

A class need not implement an entire parameterized type; instead it can implement just some of the instantiations. For example, we might have

```
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, SortedList_char might map characters to elements of a fixed-size array of integers, where each integer counts the number of occurrences of that character in the sorted list.

2.6 Optional Methods

In addition to where clauses that apply to an entire interface or class, it is possible to attach where clauses directly to individual methods, a feature originally

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

  HashBucket[Key,Value] buckets[ ]; // the hash table
  int sz; // size of the table

  public HashMap[Key,Value] (int sz) { . . . }
  // 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
}
```

Figure 3: Partial implementation of Map

```
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 4: An optional method

provided by CLU [LSAS77]. 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, neither does the instantiation. 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 4. 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.

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, SCW85]; compilers for languages with parametric polymorphism are reasonably well-understood and nothing new arises from the fact that the compiler generates bytecodes rather than machine instructions. Instead we focus on what is new: extensions to the bytecodes of the Java Virtual Machine (JVM) that are needed to express parametric polymorphism, and the effect of these extensions both on the bytecode verifier, and on the bytecode interpreter.

Extensions to the JVM are not absolutely required. The compiler could generate bytecodes for parameterized classes as though all parameters were the class Object. When compiling code that used a parameterized class, the compiler would generate runtime casts as appropriate. However, these runtime casts have an associated performance penalty, since the a dynamic type check must be performed. The relative cost of this type check can be expected to increase as more sophisticated bytecode execution technology is used. Therefore, it made sense to extend the bytecode format to be able to express parameterized types.

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 [Sun95b]. We extended the JVM in a backwards-compatible way, as described in Section 3.1. 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 3.2 and how to interpret them in Section 3.3. 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 with little performance penalty compared to the original interpreter, and without sacrificing safety. Some preliminary performance results are given in Section 3.4.

As with the rest of our design, few changes are required; our extended specification is backward compatible, allowing existing binaries (.class files) to run without modification (a minor version check is required).

3.1 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 although they do not involve a method dispatch. For example, the expression b.key.equals(k) in Figure 3 is implemented using invokewhere, as

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

Figure 5: Calling b.key.equals(k)

shown in Figure 5.

Other minor changes were made to the format of a .class file. 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 5 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.) Full details of the extensions to the JVM are provided in the appendix.

3.2 Verifier Extensions

The Java Virtual Machine 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.

The popularity of Java and suitability for WWW use derives partly from the ability to statically type-check compiled code, and it is important that the extensions for parameterized types not remove this ability. 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 needs to be verified only once. It is verified 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 class or interface; examining the bytecodes in the .class file is unnecessary.

Typechecking the bytecodes is similar to the checking performed by the compiler. If the class is parameterized, the formal parameter types are treated as ordinary types during verification, although they are really placeholders for the actual parameters. A few operations can be performed on values of these parameter types: calling operations described by the where clauses, and runtime type discrimination, i.e., casting. Parameterized code may use instantiations of other parameterized types, and these types may be instantiated either on the parameters of the current class or on ordinary types. For example, the code of HashMap[Key, Value] might mention HashBucket[K, V]. When the verifier obtains the signature of HashBucket methods, it must substitute K $\rightarrow$ Key, V $\rightarrow$ Value.

The important difference between the compiler and the verifier is that compiler variables have declared types, but the types of stack locations and local variable slots must be inferred. 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 is a procedure to merge a set of types, producing the most specific supertype. 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.

For example, consider the following Java code, which places values from two different classes in the variable x. The expression e is a boolean expression that is not of interest.

```
x = new A();
while (e) {
    x.m ();
    x = new B();
}
```

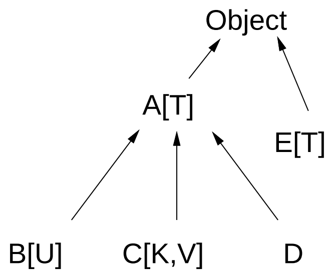
Assume that m is a method of class C (the bytecode that invokes the method indicates the class). Whether this code type-checks depends on what the types A, B, and C are. The .class file does not say what the types of local variable slots are, so the verifier must infer them. For example, the local variable slot containing variable x has an unknown type. When the method m is invoked, the stack contains the contents of x, which is either an A or a B, so the verifier finds the lowest supertype of A and B in the hierarchy. For the invocation to type-check, the lowest supertype S must be a subtype of C. Considering types as sets of legal values, S is a conservative approximation to the union of the types A and B, i.e., $A \cup B \subseteq S$. If S is a subtype of C, then $S \subseteq C$. Transitively, all possible values for x must also be legal values of type C.

The primary change to the verifier for parameterized types is a modification to this merge procedure, which must now be able to merge instantiations of parameterized classes and interfaces. This merging requires a simple extension of the obvious non-parameterized algorithm, which finds the lowest class in the hierarchy that is a superclass of all arguments.

To find the lowest common class in the hierarchy, one walks up the hierarchy from all the classes to be merged, but applying the parameter substitutions that are described by the extends clause of the class declaration. When a common class is reached in the hierarchy, the actual parameters of the class must be unified for all the merged classes, which requires that the actual parameters be equal. If we allowed within-the-bracket subtyping, then the unification rule would be relaxed.

Consider the class and interface hierarchy shown in Figure 6, with the corresponding extends clauses. The union of $B[X]$ and $C[X, Y]$ can be conservatively approximated as follows, successively moving up the tree to find a common node while substituting parameters:

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



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

Figure 6: A parameterized class hierarchy

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. 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:

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

The previous discussion had considered only parameterized classes. In the presence of interfaces, the lowest common supertype may be ill-defined, since nodes in the hierarchy may have multiple parents. However, the current Java verifier avoids this problem by deferring all checking of *interface* method calls until runtime, and we have also followed this approach.

3.3 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. Parameterized code runs about as fast as non-parameterized code, without penalizing non-parameterized code. Sun's Java interpreter makes extensive use of self-modifying code to speed up execution and perform dynamic linking lazily; we extend this technique for parameterized types.

3.3.1 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 5 contains some examples of this notation, such as the field signature <Field HashBucket[Key,Value].key Key>.

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.

For example, the HashMap method lookup from Figure 3 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.

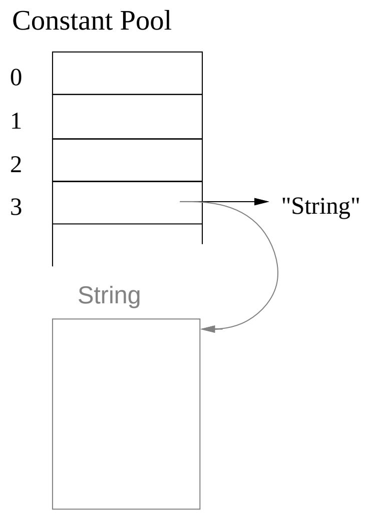


Figure 7: Resolving a constant pool entry

3.3.2 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 [Sun95b]. These quick instructions are not part of the bytecode specification and are not visible outside of the implementation. The non-quick version of these instructions 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 typical method invocation makes use of the constant pool to find the correct method to invoke, as shown in Figure 7. The figure shows some actions of the instruction new <String>, which creates a new object of class String. Suppose this instruction stores its argument at constant-pool index 3, so it is really the new 3 opcode with a constant-pool index 3 as its argument. When first executed, the instruction looks in the contents of slot 3 of the constant pool to find the name of the class, stored as a string. The name is resolved to the corresponding class object (possibly dynamically loading the class), which is then stored directly into slot 3 of the constant pool. Finally, the opcode is changed to new_quick 3, a quick instruction that will assume the constant pool entry is already resolved.

This technique will not work for parameterized classes. Consider using new with the class HashMap[T, String], in code that is parameterized on T. Each instantiation of the code has a different T, so the new should do different work for each. Our technique, shown in Figure 8, is to place this instantiation-dependent data into the ipool when the instruction (e.g., new 6) is first encountered for a particular instantiation.

In the figure, constant-pool entry 6 holds the string “HashMap[T, String]”, which is shared among all instantiations. To execute this instruction, we look at the constant-pool slot to see whether it contains the index of an ipool slot. The ipool index, 2 in this example, is assigned when the first instantiation of class HashMap is created; all instantiations use the same ipool slot index, but store different data in the slot. We look at the ipool of the instantiation running at the moment to see whether it has been initialized; if not, we fill in the entire ipool for this instantiation. Resolving ipool slot 2 will require that the instantiated value of the parameter T is substituted into the class name (producing “HashMap[Thread, String]”). The new string is resolved into an instantiation class object, which is stored in ipool slot 2. The opcode new 6 is changed to new_pquick 2, which knows to load the class from ipool slot 2. To fill in their ipools, other instantiations will need the value of the string stored at constant pool slot 6, so both the forwarding pointer and the original value are kept in that slot.[2]

In general, code that does not need the ipool is transformed using the usual quick bytecodes. The non-quick versions of certain instructions not only do a one-time initialization, but also modify the code according to whether the parameter pool must be used. This technique creates only a small one-time penalty for code that does not make use of parameterization, adding negligible overhead.

The presence of subclasses makes ipools substantially more complicated than implementing parametric polymorphism for non-object-oriented languages like CLU or ML [MTH90]. 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. However, the ipool to use for the call is not necessarily the ipool of the receiver's class, since the method may be inherited from another class. An additional indirection is required to obtain the correct ipool, but this is not too expensive.

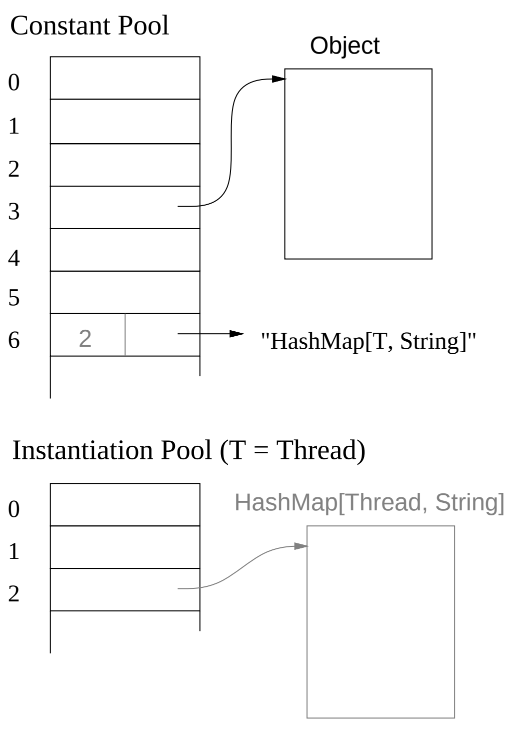


Figure 8: Resolving an ipool entry

Our implementation technique resembles an earlier approach [DGLM95], but also provides support for static methods, static variables, and dynamic type discrimination. It is applicable to compiled machine code as well as the JVM bytecodes.

3.3.3 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. Large primitive types such as long and double,

[2]For some constants, storing the forwarding pointer requires an additional constant pool slot. This is discussed more fully in Appendix B.

Original interpreter:	8.5
Extended interpreter:	8.7

Figure 9: Java Compiler Speed

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

Figure 10: Collection Class Results

which typically take up more space than object pointers, can be handled in several ways. The most efficient technique is probably for the class loader to instantiate the parameterized class for those particular types by rewriting bytecodes. If a parameterized class has an instance variable of type T, and is instantiated with T = long, the field offsets of other instance variables may change for that instantiation, since the long will take up more space than other parameter types. This technique does duplicate code, however.

3.4 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. The results of our benchmarks are shown in Figures 9 and 10. 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 9, the extended interpreter runs only 4% 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 identical, except for changes in type declarations and a dynamic cast.

invokewhere	22.8
invokevirtual	21.9
invokeinterface	23.3

Figure 11: Invocation Speed

In the first micro-benchmark, we compared the parameterized collection class to the same collection class where the types of the components are hard-wired. We would expect that the hard-wired class, though less generally useful, would run faster. As Figure 10 shows, there is only a 4% penalty for running parameterized code.

In the second micro-benchmark, we compared the parameterized collection to the same collection class where the types of the components are all Object. 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. The existing Java utility classes mostly follow this philosophy. In 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. This experiment represented a best case for invokeinterface, where its inline caching worked perfectly. In our results, all three method invocation paths are approximately the same speed.

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

We have already mentioned in Section 2 that in order for primitive types such as int to be used in conjunction with parameterization (or at least to satisfy where clauses), it is necessary for them to have methods. Therefore, we have assumed that they do indeed have methods that correspond to the abilities provided by the infix notation. Thus, integers have all the common comparison and arithmetic methods, etc. As mentioned, this does not mean that these methods can be called directly (instead of using infix notation) in non-parameterized code, although in fact we think it would be a good idea to allow this. Also, the fact that the methods exist does not rule out the use of efficient implementations, e.g., x + y can still be implemented with the iadd bytecode (and if x.add(y) were allowed, it also could be implemented with this bytecode). Of course, inside a parameterized class, t1.add(t2), where the type of t1 and t2 is the parameter type T, a call to the where-routine must be made (except that one could macro-expand a particular instantiation, e.g., when the parameter is int, and replace the call with iadd.)

In addition, it is important to use common naming conventions for the methods of the primitive types. For example, at present Java strings have a compare method (which does a three-way comparison of its argument with this). A method like this is fine, but either all primitive types should have such a method, or all primitive types should have the various comparison methods (lt, equals, etc.). 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 Renaming

More complicated instantiation mechanisms have been proposed that would eliminate the dependance on names discussed above. In Argus [LDH$^+$87] the instantiation can select the routine to bind to the where-routine, e.g., you could say SortedList[Num{p for lt}] to substitute some routine p for the lt method. Such a mechanism provides more expressive power but is more complex and difficult to explain than what we propose here. Also, it becomes difficult to decide what the right definition of type equality ought to be. For example, is SortedList[Num] = SortedList[Num{p for lt}]? Presumably the answer should be yes if p behaves the same as lt but not otherwise. Since this is not a question that a compiler can answer, Argus takes the conservative approach of considering the types equal only if p and int.lt are the very same code. This provides a bit more expressive power than our where clauses, but in a language like Java where code is loaded dynamically it means that more type checking must be deferred until runtime. Therefore we have not provided this ability.

Another possible approach is to avoid where clauses and instead pass the needed routines in 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). This requires first-class procedures, which can be passed as arguments and results, and stored in data structures. Java does not have first-class procedures at present (although we assume this shortcoming will be addressed shortly). Furthermore, there must be a way to associate procedures with methods (so that the equivalent of int.lt could be passed).

However, even assuming such mechanisms exist, passing procedures to constructors is not a good solution. The problem is that 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 Map[Foo,Bar] objects would behave very differently if different equality tests were used inside them. Our approach prevents such objects from being considered as belonging to the same type, by making it impossible to have more than one Map[Foo,Bar] type. (Renaming would also keep the types separate, but would allow there to be more than one Map[Foo,Bar] type. This feature would require of runtime type checking.)

4.3 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 2 for parameterized classes and interfaces.

The subtyping rule for Java arrays *does* allow subtyping covariant in the parameter types. Thus, 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.[3]

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

Furthermore, the semantics of the Java approach are unfortunate from a methodological point of view. A subtype's objects ought to behave like those of the supertype so that people writing programs in terms of the supertype can reason about their behavior using the supertype specification [LW94]. Java arrays violate this rule.

Therefore, we believe that for general parameterization of classes and interfaces it is better to not 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 it would be good if the Java rule for arrays were changed so that all parameterized types had the same rule. However, our design does not require this; different rules could be used for arrays and for other parameterized types.

4.4 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 much more useful if parameterization had been available.

For example, consider the type Vector, which is an extensible array. Because of the lack of parameterization, all elements in a Vector must considered to be of type Object, even in a context where their type is much more narrow. This has two unfortunate consequences:

```
interface Enumeration[T] {

    bool hasMoreElements ( );
    // effects:  returns true if there are more
    //           elements to yield in this

    T nextElement( ) throws NoSuchElementException;
    // effects:  if there are more elements to yield,
    //           yields one and records the yield,
    //           else throws NoSuchElementException

}
```

Figure 12: Specification of parameterized Enumeration type

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 use class *wrappers* to *objectify* them.

2. Whenever an object is fetched from a vector, it must be cast to the expected type.

Note that both wrapping and casting have a runtime cost; in addition they are ugly and make code less convenient to write.

The problems can be completely eliminated if Vector is parameterized. Then a Vector[int] is known to contain ints and there is no need to either wrap the int going in or cast it coming out. Furthermore, note that you can also have Vector[Object] in the case where what you want is a heterogeneous vector, so parameterized vectors have all the necessary expressive power.

Another example is the Enumeration interface, which attempts to capture the notion of a generator that provides a sequence of values. Like vectors, enumerations are heterogeneous, so that the elements produced must be cast to the expected type before use. A much better interface can be achieved by using parameters; such an interface is shown in Figure 12.

Every type that provides an ability to enumerate over its elements simply provides an implementation this interface, or of a specific instantiation of this interface. For example, SortedList might have the method shown in Figure 13.

However, the elements method in a class that implements SortedList[int] would return Enumeration[int] as shown in Figure 14. (Both classes in the figure are in

[3]The check can be avoided if the compiler can figure out what is going on at compile time. Also, the check is only needed when T has subtypes. Both primitive types and final classes satisfy this property, but Sun's current Java interpreter only removes the type check for the primitive type case.

```
        Enumeration[T] elements ( );
        // effects: returns an enumeration object that
        //          can be used to obtain the elements
        //          of this in sorted order
```

Figure 13: The elements iterator

```
class Intlist implements SortedList[int] {

        int [ ] els; // keep elements in a sorted array
        ...

        Enumeration[int] elements ( ) {
                return IntlistGen(this);
        }
}

class IntlistGen implements Enumeration[int] {

        Intlist li; // the list
        int pos;  // position of last element yielded

        IntlistGen(Intlist x) {
                li = x;
                pos = −1;
        }

        int nextElement( )
           throws NoSuchElementException {
                if (pos ≥ els.high)
                    throw new NoSuchElementException();
                pos++;
                return els[pos];
        }

}
```

Figure 14: Using parameterized enumerations

the same module, and therefore the IntlistGen class has access to the representation of Intlist objects.)

5 Conclusions

Java offers for the first time 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, as discussed in Section 4.

The paper also explains how to implement the extensions. The implementation has three main goals: to allow all instantiations to share the same bytecodes (thus 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. That section describes a few new bytecodes that are needed to support parameterized abstractions; it also described the designs of the bytecode verifier and interpreter, and the runtime structures they rely on.

Our bytecode interpreter has been implemented (except for static variables in parameterized classes). Our preliminary performance results, in Section 3.4, show that we have roughly a 4% penalty for the presence of parameterized code, but that some common code is sped up by 16% because parameterized code can avoid some runtime type checks. We expect that some simple performance tuning can improve these results.

Parameterized abstractions are not the only extension needed to make Java into a convenient general purpose programming language. Another needed extension is first-class procedures; we discussed them briefly in Section 4.2. This paper is not concerned with providing a full analysis of missing features, but we would like to close with a plea for *iterators*. Enumeration over elements of collections can be provided using generators, as discussed in Section 4.4, but it is much more convenient to use iterators. Iterators were first introduced in CLU; they are routines similar to procedures, except that they yield a sequence of results, one at a time; see [LSAS77, LG86] for discussion and examples. For example, the elements method in Figure 14 becomes:

```
    int iter elements ( ) {
        for (int i = 0; i < els.high; i++) yield els[i];
    }
```

This is much simpler than writing the generator; rather than needing to provide a class for each enumeration, all that is needed is just a simple routine. And the difference is even more pronounced when the data structures are more complex (e.g., recursive structures such as trees and graphs).

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:

$$\begin{array}{rcl}
\textit{interface} & \rightarrow & \left[\textit{InterfaceModifiers}\right] \textbf{ interface } \textit{idn} \\
 & & \left[\,[\ \textit{params}\ ]\,\right]\ \left[\textit{where}\right] \\
 & & \left[\textit{ExtendsInterfaces}\right]\ \textit{InterfaceBody} \\
\textit{params} & \rightarrow & \textit{idn}\ \left[\textbf{,}\ \textit{idn}\right]* \\
\textit{where} & \rightarrow & \textbf{where}\ \textit{constraint}\ \left[\textbf{,}\ \textit{constraint}\right]* \\
\textit{constraint} & \rightarrow & \textit{idn}\ \{\ \left[\textit{whereDecl}\ \textbf{;}\right]*\ \} \\
\textit{whereDecl} & \rightarrow & \textit{methodSig} \mid \textit{constructorSig} \\
\textit{methodSig} & \rightarrow & \left[\ \textbf{static}\ \right]\ \textit{ResultType} \\
 & & \textit{MethodDeclarator}\ \left[\textit{Throws}\right] \\
\textit{constructorSig} & \rightarrow & \textit{ConstructorDeclarator}\ \left[\textit{Throws}\right]
\end{array}$$

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

$$\textit{instantiationType} \rightarrow \textit{idn}\ [\ \textit{actualParams}\ ]$$

where

$$\textit{actualParams} \rightarrow \textit{Type}\ \left[\textbf{,}\ \textit{Type}\right]*$$

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:

$$\begin{array}{rcl}
\textit{ClassDeclaration} & \rightarrow & \left[\textit{ClassModifiers}\right] \textbf{ class } \textit{idn} \\
 & & \left[\,[\ \textit{params}\ ]\,\right]\ \left[\textit{where}\right]\ \left[\textit{Super}\right] \\
 & & \left[\textit{Interfaces}\right]\ \textit{ClassBody}
\end{array}$$

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:

$$\begin{aligned} \textit{MethodDeclaration} \quad \rightarrow \quad & \left[\textit{MethodModifiers}\right] \textit{ResultType} \\ & \textit{MethodDeclarator} \left[\textit{Throws}\right] \\ & \left[\textit{where}\right] \textit{MethodBody} \end{aligned}$$

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 types double and long cannot be used as type parameters at the VM level. The Java compiler will 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 5.

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:

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

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173*, pages 51–68. Springer-Verlag, 1984.

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of OOPSLA '95*, Austin TX, October 1995. Also available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/where-clauses.ps.gz.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[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. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LDH$^+$87] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. *Argus Reference Manual*. Technical Report 400, MIT Laboratory for Computer Science, November 1987.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *CACM*, 20(8):564–576, August 1977.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[MTH90] Robin Milner, Mads 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 *OOPSLA '95 Conference Proceedings*, Austin, TX, October 1995. Also available at ftp://ftp.pmg/pub/thor/bidirectional.ps.gz.

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

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment, Language Reference Manual*. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985. Published as *SIGPLAN Notices 21(11)*, November, 1986.

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

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

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

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