Adding Type Parameterization to the Java™ Language

Ole Agesen
Sun Microsystems Laboratories
2 Elizabeth Drive
Chelmsford, MA 01824
ole.agesen@Sun.COM

Stephen N. Freund* John C. Mitchell*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu

Abstract

Although the Java programming language has achieved widespread acceptance, one feature that seems sorely missed is the ability to use type parameters (as in Ada generics, C++ templates, and ML polymorphic functions or data types) to allow a general concept to be instantiated to one or more specific types. In this paper, we propose parameterized classes and interfaces in which the type parameter may be constrained to either implement a given interface or extend a given class. This design allows the body of a parameterized class to refer to methods on objects of the parameter type, without introducing any new type relations into the language. We show that these Java extensions may be implemented by expanding parameterized classes at class load time, without any extension or modification to existing Java bytecode, verifier or bytecode interpreter.

1 Introduction

In Ada generics [US 80], C++ templates [ES90], and ML polymorphic functions and data types [Mil85, Ull94], type parameterization is used to allow a general concept to be instantiated to one or more specific types. For example, we may define a package or class implementing a generic `List` datatype, and instantiate it to produce lists of integers, lists of database records, or lists of other types of objects. As borne out by experience with these and other languages, the ability to define and instantiate a general concept is particularly useful in the design and construction of reusable libraries. To cite a specific example, the C++ Standard Template Library (STL) [MS95, PSLM96] uses C++ templates extensively.

While the current Java language does not provide any form of type parameterization, it does include a supertype `Object` of all object types. This allows some classes that could be written in other languages using parametric polymorphism to be expressed using `Object` in place of a parameter type. However, there are several disadvantages of this substitute for type parameterization. First, the programmer must insert type casts to change an expression with static type `Object` to a more specific type. This can be an annoyance and, since Java type casts are checked at run time, it also leads to some decrease in execution speed. The use of superclass `Object` becomes more cumbersome when binary operations (such as an ordering) are needed on objects of the parameter type, since typing considerations become more complex. Finally, use of a fixed type `Object` does not allow us to capture certain generic concepts that could be expressed as parameterized classes or interfaces. For all of these reasons, we believe that it is important to add type parameterization to the Java language.

This paper proposes a language design and implementation. Although we considered the language design problem from first principles, the result of our deliberation and programming experiments coincides substantially with other proposals [MBL97, OW97a, OW97b]. Our implementation approach, based on instantiation of generic classes and interfaces at class load time, differs significantly from other approaches that have been explored in any detail. This implementation technique allows us to support a more expressive form of parameterized classes than competing implementation techniques. It may also be considered an alternative implementation mechanism for related language design proposals.

The Java language extensions that we consider allow classes and interfaces to be defined using type parameters that may be instantiated with classes, interfaces, or primitive types. To enable classes to refer to the meth-

*Funded in part by NSF Grants CCR-9303099 and CCR-9629754. Stephen Freund received additional support through an NSF Graduate Research Fellowship.

To appear in OOPSLA '97.

ods of objects from a parameter type, each type parameter may be assumed to implement a given interface or extend a given class. This reuse of existing Java type relations `extends` and `implements` to constrain type parameters reduces the number of new concepts added to the language. This is essentially similar to the design used in [OW97b], also based on so-called F-bounded quantification [CCH+89, Mit96].

Our implementation technique adds a preprocess phase to the Java Virtual Machine loader. The preprocessor instantiates parameterized classes at class load time to achieve a balance between pre-run-time expansion (for expressive power and run-time efficiency) and minimization of compiled (or transmitted) code size. We have implemented a prototype version of the loader preprocessor using approximately 500 lines of Java code. While it was necessary to make one small change to the Java Virtual Machine loader in order to invoke our preprocessor at the correct time, this small change could be incorporated into future versions of the Java Virtual Machine without affecting backwards compatibility. More importantly, it is not necessary to change the verifier or bytecode interpreter. This is significant since many of the security and portability properties of Java are tied to the design of these parts of the virtual machine.

To summarize, our goals in adding type parameters to the Java language are:

- Allow common parameterized classes to be programmed easily. We are more interested in making it easy to handle common, useful cases than in maximizing absolute language expressiveness.
- Avoid introduction of extraneous concepts into the Java language, where possible.
- Support separate compilation. In keeping with the Java emphasis on security, it should be possible to check each compilation unit separately for consistency and type correctness.
- Implement the extended language with as few changes to the virtual machine as possible. In particular, avoid any changes in the Java bytecode, Java verifier, or bytecode interpreter.
- Maintain static type checking insofar as possible. In particular, eliminate the need for programmers to write casts, minimize associated run-time costs, and avoid introduction of additional constructs that would increase the use of run-time tests to preserve type safety.

We review type parameterization in other languages in Section 2 and describe four representative and increasingly difficult "benchmark" program examples. In Section 3 we summarize relevant features of the existing Java language and outline how the lack of type parameterization requires work-arounds for each of the four program examples. Then, in Section 4, we present our language extension. Section 5 presents our preprocessor-based implementation technique and describes a prototype that we have built to validate our ideas. Section 6 compares our design and implementation with related proposals, while Sections 7 and 8 outline directions for future work and conclude.

2 Parameterized Classes

Many programmers are familiar with generic program units, as supported by Ada generics [US 80], C++ templates [ES90], and ML functors, polymorphic functions, and data types [Mil85, Ull94]. In brief, each of these constructs provides some mechanism for treating a program unit as a "generic abstraction" that can be instantiated for specific choices of actual type parameters and function parameters. To give a familiar example, a generic C++ `Vector` class may be defined as follows [ES90]:

```
template <class T> class Vector {
    T* v;
    int size;
public:
    Vector(int);
    T& operator[](int);
    T& elem (int i){return v[i];}
    // ...
}
```

For any class `C`, the class `Vector<C>` is a class of vectors of `C` objects, determined by instantiating the class `Vector` with actual class `C` in place of the formal parameter `T` in the definition above. Parametric polymorphism seems to have originated with the design of CLU [L+81], which allows procedures and clusters (approximately analogous to Ada packages or ML structures) to be parameterized by types. The prevalence of type parameterization across a wide variety of programming languages provides some evidence for the general applicability and power of this concept.

2.1 Compiling Parameterized Classes

Any type parameterization mechanism will involve both a technique for compiling parameterized constructs and a mechanism or set of conventions for using specific instances of a generic construct. We discuss the general issues that arise in any language using parameterized classes as an example. However, the main points apply to parameterized interfaces or other parameterized entities as well.

In many cases, it is necessary for a parameterized class to invoke methods on objects of the parameter type. This does not occur with certain "container" classes, such as `List` or `Vector`, but arises in the implementation of priority queues, for example, since it is necessary to have an ordering on the elements inserted into a priority queue. Such method invocations raise the question of how to ensure type safety. More specifically, we must be able to ensure that methods invoked on objects whose types are parameters are actually defined on the types with which the parameters are instantiated.

Four representative examples may be used to illustrate some of the challenges involved in designing an expressive type parameterization mechanism. In order of increasing complexity, these are:

`Stack` class: Like vectors, lists, and other simple container classes, stacks of `T` objects may be implemented without invoking any methods on objects of type `T`.

`Hashtable` class: In implementing a hash table, it is necessary to obtain a hash code for each object inserted in the table. For the purposes of illustration, we assume that objects have a `hashCode` method returning an integer.

`PriorityQueue` class: This is similar to `Hashtable`, since a priority queue of `T` objects requires a method on `T` objects. However, the type of the method is more complicated, since the method must provide a binary relation that would be assumed to be a linear order.

"mix-in," a term we use for a parameterized class that uses a type parameter as a base class. An example is a generic construct that adds methods `notEqual`, `greaterThan`, `lessEqual`, and `greaterEqual` to any class with `equal` and `lessThan` methods. In a parameterized class of this form, it may be necessary to assume that certain binary methods are provided by the actual parameter class.

The increasing complexity of these examples is really a measure of the complexity of the way in which the parameterized classes use the type parameters. The `Stack` class performs no operations on instances of the parameter class and does not need to make any assumptions about the type parameter. The `Hashtable` and `PriorityQueue` classes invoke methods on instances of a type parameter class and therefore must ensure that these methods are defined and have the appropriate types. Finally, mix-ins, the most complex example, use the type parameter as a superclass. This will generally require information about the methods and constructors associated with the type parameter.

All of these examples may be programmed in our extension of the Java language. However, the last one is not possible in some of the competing proposals described in Section 6. There are also differences regarding the use of constructors and static variables, as discussed in Section 2.2 below and in more detail in Section 6.

2.2 Instances of Generic Classes

Once we have defined a generic construct, such as a parameterized `List` class, we may wish to write a program with several specific types of lists, such as lists of integers and lists of employee record objects. We refer to each class name, such as `List<int>`, `List<Emp>`, or `List<Color>`, obtained by specifying an actual type parameter, as a *syntactic instance* of the parameterized class `List`. In any implementation of generic classes, it will be necessary to decide how to represent objects of type `List<int>`, `List<Emp>`, and `List<Color>`, and whether to generate separate code for methods associated with objects of these types.

In some languages, it may be possible to use the same implementation (code and static variables) for all syntactic instances. For example, if list cells are represented appropriately, with pointers to data (instead of the data itself) in each cell, then it may be possible to use the same representation and compiled operations for all types of lists. This is done in ML, for example. We refer to the situation where all instances of a parameterized construct share the same implementation as a *homogeneous* approach, following the terminology of [OW97b].

Alternatively, parameterized classes could be implemented *heterogeneously*, with different syntactic instances resulting in different run-time representations or different compiled code for methods. This is generally the case for C++, for example. In a heterogeneous implementation, we refer to the data structure and compiled code associated with a specific syntactic instance of a generic class as a *semantic instance*, or *instantiation*, of the class. In general, semantic instances might be *instantiated* at compile time, link time, or run time. While a heterogeneous implementation may result in larger code size, a heterogeneous strategy makes it possible to support type parameterization in a language where it is not feasible to represent or refer to values of all types in a uniform way.

Since it is related to some of the subsidiary design decisions we discuss in this paper, we note that there are some interactions between compilation of generic classes and the eventual creation of semantic instances. For example, C++ templates are compiled without statically checking the use of member functions, in part because

```java
class Stack {
  void push(Object o)  { ... }
  Object pop() { ... }
  ...
}

String s = "Hello";
Stack st = new Stack();
st.push(s);
...
s = (String)st.pop();
```

```java
class Stack<A> {
  void push(A a) { ... }
  A pop() { ... }
  ...
}

String s = "Hello";
Stack<String> st = new Stack<String>();
st.push(s);
...
s = st.pop();
```

Figure 1: Java code for a generic `Stack` class: versions without and with parameterized classes.

of the complexity of determining in advance how a reference to a function might be eventually resolved. Instead, type checks are made later, at link time, when the actual type parameter is determined and a semantic instance is created.

The choice between homogeneous and heterogeneous implementation strategies is often determined by language semantics. For example, consider a parameterized class that defines a static variable. If the semantics specify that each instance of the parametric class gets its own location for the static variable, it may be more straightforward to use a heterogeneous implementation strategy. On the other hand, if the semantics specify that all instances should share a single location for the static variable, a homogeneous implementation strategy may be a better choice (with respect to this particular aspect of parameterized classes).

3 The Java Programming Language

Since the Java language has received considerable attention, we assume that most readers have at least a passing acquaintance with it. The main features of the Java language that are relevant to our proposal are: the presence of interfaces and classes, the absence of type parameterization, the fact that type casts are checked at run time (to ensure type security), and, for our implementation, the architecture of the Java Virtual Machine. These features are documented in standard Java language reference books such as [AG96]. We also review the architecture of the Java Virtual Machine briefly in Section 5 and summarize other features in passing as needed.

In the following subsections, we discuss alternative programming techniques that may be used to work around the lack of type parameterization when solving the four representative problems from Section 2.1. The purpose of this discussion is to motivate our specific combination of language extension and implementation technique, as well as to provide a basis for comparison with various alternatives. In particular, the first three examples may be written in the other extensions described in Section 6. However, the last example, which is expressible in our extenstion, cannot be written and implemented properly by some of the other proposals. The main difference here is not the form of the language extension, but our heterogeneous implementation technique.

3.1 Stack

Without type parameterization, it is possible to write a Java `Stack` class using the fact that all classes are subclasses of `Object`. A typical approach is shown on the left in Figure 1.

This `Stack` class defines two essential operations: `push` places an object on the stack, and `pop` removes and returns the top object. The type `Object` occurs in two places: for the argument of the `push` method and for the result of the `pop` method. While the type of `push` is very generous, allowing any type of object to be added to a stack, a price is paid when the object is removed. Specifically, since the return type of the `pop` method is just `Object`, it is only possible to invoke methods from class `Object` on an object removed from a stack, even though the object may have additional methods. It is possible to cast an object removed from a stack to a more specific type, but this action requires either that the type be known to the programmer or that some test be performed to determine the type. In addition, to ensure type safety, the Java compiler inserts a run-time test with such a cast, causing some performance penalty.

In the best-case scenario for the current Java language, a programmer really wants heterogeneous stacks containing many types of objects and, before invoking certain methods, must cast objects removed from stacks to more specific types.

In the worst-case scenario, however, a programmer defines a stack class as above, and builds stacks with every element drawn from some specific type, such as `String`. Then, in spite of the fact that the program only builds homogeneous stacks, it must contain casts and run-time type tests to verify something already known to the programmer. These casts can be avoided by using a parameterized class, as shown on the right of Figure 1.

3.2 Hashtable

To define a general Java `Hashtable` class without type parameterization, we define an interface `Hashable` which asserts that objects with this interface have a `hashCode` method. Then, to use objects of some type as keys for a hash table, the class of these objects must explicitly be declared to implement the `Hashable` interface. Moreover, the cast issues described above for stacks remain for `Hashtable`. This approach can be compared with the parameterized hash table class described in Section 4 and given in Figure 2.

3.3 Priority Queue

A priority queue is a data structure that allows insertion and removal of objects of some type `T`. The objects must come from a linearly ordered set so that removal may return the minimum of all the elements stored in the data structure. The ordering, which we assume is provided by a `lessThan` method, poses certain problems in writing a generic priority queue class. A specific complication is that the `lessThan` method on objects of type `T` takes another object of type `T` as an argument. However, this type dependence cannot be expressed in Java without type parameterization. It is possible to work around the problem by assuming that each object to be inserted into the queue has a `lessThan` method that accepts any object as a parameter. However, the work-around sacrifices compile-time type checking and is cumbersome: the `lessThan` method must perform a run-time type check to make sure that the argument actually has the correct type. (See [BCC+96] for further discussion.) We demonstrate how a priority queue may be written with our language extension after we present the required syntax in Section 4.

3.4 Mix-in

For examples such as a generic "mix-in" class that adds operations to a class provided as a parameter, there does not appear to be any reasonable approach short of parameterized classes. Figure 4 in Section 4 demonstrates a generic mix-in class.

4 Language Design

4.1 Java Language Extensions

In our extension to the language, parameterized classes have the following general form:

```
class C< parameters > {
   ...
}
```

where *parameters* is a list of type variables, each of which may be constrained or unconstrained. Interfaces can be parameterized in the same way. The type variables, which represent unknown classes, interfaces, or primitive types, may be used inside the declaration of `C` wherever a type name is allowed to appear.

Constraints on type variables have two forms, **`implements`** and **`extends`** clauses. These concepts, already familiar to Java programmers, have the following intended meanings:

- `A implements I`: The parameter `A` is either a class which implements `I` or an interface which has `I` as a superinterface.
- `A extends B`: The parameter `A` is a class which is derived from `B` in the implementation hierarchy.

A similar syntax is used in [OW97b], where a BNF grammar, formal typing rules, and other syntactic issues are considered. The main difference between our proposed language extension and Pizza lies the way that our heterogeneous implementation may be used to support certain uses of type variables that Pizza disallows. These differences are discussed in Section 6.

Figure 2 demonstrates the use of constraints to specify that the `Key` parameter for `Hashtable` must implement a certain interface. The `Key` argument is constrained to only those classes that implement the `Hashable` interface, ensuring that the `Hashtable` code can invoke the `hashCode` operation on the keys. In contrast, no specific operations are required on the values inserted in the hash table, so the `Value` parameter is unconstrained, and any type will suffice.

In general, a parameterized class may be statically type checked by the compiler using the constraint information about the parameters. However, there are several situations in which the constraints do not express enough information to guarantee that all possible instantiations will conform to all the existing

```
interface Hashable {
  int hashCode();
}

class Hashtable<Key implements Hashable, Value> {
  void insert(Key k, Value v) {
    int bucket = k.hashCode();
    insertAt(bucket, k, v);
  }
  void insertAt(int bucket, Key k, Value v) { ... }
  ...
}

class Name implements Hashable {
  int hashCode() { ... }
  ...
}

Hashtable<Name, Integer> h1 = new Hashtable<Name, Integer>();
Hashtable<Name, Point> h2 = new Hashtable<Name, Point>();
Name n = new Name("Hello");
h1.insert(n, new Integer(2));
h2.insert(n, new Point(0));
```

Figure 2: A parameterized `Hashtable` class and instantiations.

rules of the Java language. For example, when a type variable is constrained to extend a class, the actual parameter must be a class since Java interfaces cannot extend classes. In the following code example,

```
class C<A> extends A {
  void f() {
    A a = new A();
  }
}
```

the actual parameter `A` must be a class because it is used as a base class. In this case, `A` must also be a class because the method `f` creates an object of type `A`. Similarly, since the extension of an interface must be an interface, the following code,

```
interface I<B implements J> extends B {
  ...
}
```

implies that `B` is an interface with superinterface `J`.

The Java typing rule for method overloading creates another similar restriction. Consider the following class declaration:

```
class L<A> {
  void m(A s)      { ... }
  void m(String s) { ... }
}
```

The class overloads the method name `m` to operate either on a string or an instance of the formal type parameter `A`. In this situation, the instantiation `L<String>` must be rejected because it leads to a class defining two methods with the same signature `m(String)`.

In practice, situations that impose such additional restrictions on actual type parameters occur rarely. They can be statically determined and checked for each instantiation by the compiler. We do not expect them to be difficult for programmers to understand because they all arise from the unsurprising rule that "an instantiation of a parameterized class must conform to all the rules for regular (non-parameterized) classes."

4.2 Discussion

This type parameterization mechanism meets our primary language design goals. As previously mentioned, the **extends** and **implements** constraints are based on concepts already present in the Java language. This should make these constraints readily understandable and easy to use by any Java programmer.

```
interface Comparable<I> {
  boolean lessThan(I);
}

class PriorityQueue<T implements Comparable<T>> {
  T queue[];
  void insert(T t) {
    ...
    if (t.lessThan(queue[i])) ...
    ...
  }
  T remove() { ... }
  ...
}
```

Figure 3: A parameterized `PriorityQueue` demonstrating the declaration and use of a contravariant method.

```
interface LessAndEqual<I> {
  boolean lessThan(I);
  boolean equal(I);
}

class Relations<C implements LessAndEqual<C>> extends C {
  boolean greaterThan(Relations<C> a) {
    return a.lessThan(this);
  }
  boolean greaterEqual(Relations<C> a) {
    return greaterThan(a) || equal(a);
  }
  boolean notEqual(Relations<C> a) { ... }
  boolean lessEqual(Relations<C> a) { ... }
  ...
}
```

Figure 4: A mix-in class to add additional relations to any class defining `lessThan` and `equal` relations.

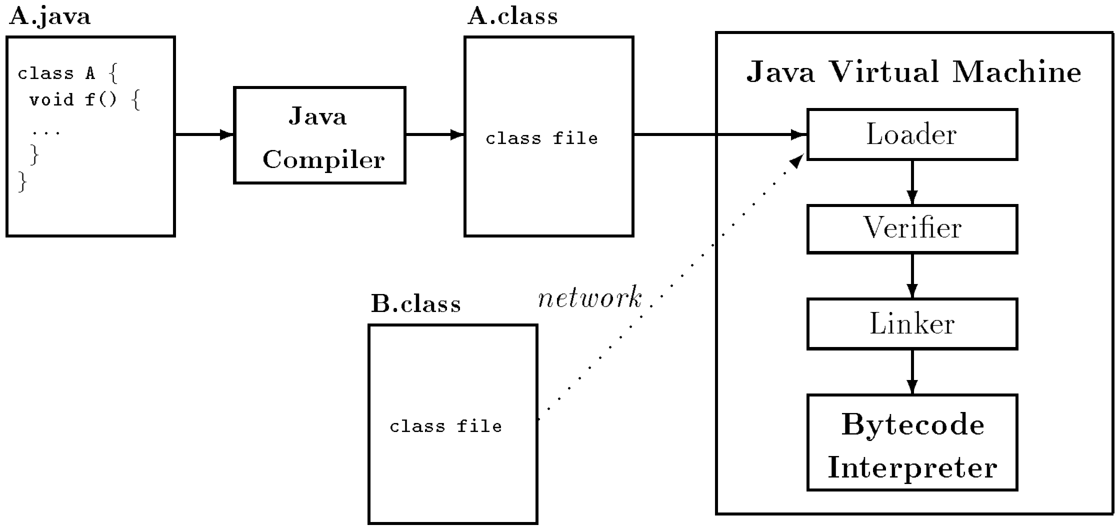

Figure 5: The Java Virtual Machine

The parameterization mechanism is also quite expressive. In addition to the examples already discussed, contravariant method constraints can be described. (A contravariant method is one which takes an argument of the same type as the class in which the method is defined; see [Cas95] for a detailed discussion.) The binary `lessThan` method of the priority queue described in Section 3.3 is a contravariant method. Figure 3 shows how this type of relation may be expressed in our language extension. An even more complex example is shown in Figure 4. The mix-in class defined in the figure not only specifies the existence of contravariant methods for its parameter type, but it also inherits from the parameter.

Parameterized classes and interfaces can be statically type checked using the constraints associated with the parameters. This kind of static checking has been implemented in the Pizza compiler [OW97b] and also the Rapide compiler [KLM94], now in use for several years.

4.3 Primitive Types

Our design is sufficiently flexible to allow instantiation with primitive types such as `int` and `float`. However, since primitive types do not implement or extend any classes or interfaces, they can only be used for unconstrained parameters. Also, the resulting instantiations must adhere to all Java type rules relating to primitive types. For example, the class `Stack<int>` is valid, but `Hashtable<int, float>` is not since the primitive type `int` does not implement the `Hashable` interface. While allowing instantiation with primitive types enhances expressiveness, it, comes at a cost in implementation complexity, as discussed in Section 5.3.

To allow basic types to be used where class types are required, the Java class library defines "wrapper" classes for primitive types [AG96, §13.3]. These wrappers can also allow primitive types to be used as actual parameters to classes that only allow instantiation with class or interface types.

4.4 Extended Constraints

While the syntax used in this paper allows only one constraint per type parameter, it is essentially straightforward to allow combinations of constraints. For example, we could use the form

```
class C<A implements I extends B>
```

to require `A` to both implement interface `I` and extend class `B`. For longer constraints, it seems more convenient to write constraints after the parameter list, as is customary with `where` clauses [LSAS77].

5 Implementation

To set the context for a discussion of our implementation technique, Figure 5 shows a schematic diagram of the Java Virtual Machine. When a running program refers to a class that has not been loaded, compiled bytecode passes through three largely independent components of the virtual machine. The *loader* obtains the bytecode for the class from either a local file or a remote site. The *verifier* validates the bytecode by checking that operation codes are valid, branches are to legitimate locations, methods have structurally correct signatures, and that every instruction obeys the Java type discipline. Finally, the *linker* initializes static fields of the new class and may load related classes. Further information may be found in [LY96].

It is possible for Java programs to modify the behavior of the loader on certain classes by extending the Java `ClassLoader` class. For example, while the default loading mechanism searches the local file system for classes, an alternate `ClassLoader` could obtain classes over the network.

Parameterized classes can be implemented without changing the Java Virtual Machine. Homogeneous implementations, such as those described in [OW97b, Tho97], represent a type variable as a general type, such as `Object`, and use run-time casts to provide the appropriate functionality for different instantiations. Some limitations of this strategy are discussed in more detail in Section 6.

In order to overcome these limitations, we propose a heterogeneous implementation based on *load*-time expansion. This choice is attractive for Java systems for the following reasons:

- Expanding parameterized classes to a form used by non-parameterized classes allows current Java Virtual Machine verifiers, interpreters, and just-in-time compilers to remain unchanged. In addition to retaining compatibility, we also avoid security and other issues that would arise if we changed the Java execution model.
- Postponing expansion from compile time to load time reduces the size of compiled code. This could be significant when compiled code is transmitted over a network.

The general strategy is to compile a parameterized class into an extended form of the class file. In addition to all the information usually found in a class file, the extended file includes information about parameters and constraints. When the virtual machine attempts to load an instantiation of a parameterized class or interface, a loader preprocess phase transforms the parameterized class file into the desired instantiation and then declares it as if it were a normal class. In other words, one "template" class file is used to generate regular, non-parameterized `Class` objects for each instantiation of a parameterized class. For example, the contents of the `Hashtable.class` file is used by the loader to generate both the `Hashtable<Name, Integer>` and `Hashtable<Name, Point>` classes.

The rest of this section describes our implementation technique in more detail. Section 5.1 outlines the Java class file format. Section 5.2 describes our preprocessor, focusing on the transformation used to instantiate a parameterized class with a class or an interface. Then, Section 5.3 sketches instantiation with primitive types. Section 5.4 discusses advantages and disadvantages of this implementation technique, and Section 5.5 describes a prototype implementation and our experience with it.

5.1 Java Class Files

A class file is a binary representation for a compiled class that can be loaded by any Java Virtual Machine. The major pieces of the class file are a header, starting with a distinguishing magic number, a constant pool, and a description of the fields and methods in the class. This format is summarized in Figure 6 and discussed in detail in [LY96].

The constant pool contains strings representing all names mentioned in the class file, including class names, field and method names, type names, and so on. The index of an entry in the constant pool is used wherever that constant is needed in the class file. The Java Virtual Machine uses these strings for many operations including type checking, dynamic linking, and run-time resolution of field and method names. For example, Figure 7 shows some Java bytecodes and the corresponding program text. The `#4`, `#5`, and `#6` in the bytecodes refer to entries in the constant pool, and the text printed after these numbers are the strings stored in the constant pool.

5.2 Load-Time Expansion

One very convenient aspect of the class file format is that most type information generated when a class is compiled is stored in its constant pool as strings. The only exception is type information about primitive types, such as integers, which is embedded directly into the bytecode instructions (see Section 5.3).

To support load-time instantiation of parameterized class files, we propose an extended class file format, as shown in Figure 6. The file begins with a unique identifying number different from the standard class file magic number. Following this are the argument constraints, whose form is also shown. The format of the constraints is straightforward. The one constraint generated by our `Hashtable` example is:

1
implements
"Hashable"

If we were to allow multiple constraints on each type parameter, the constraint format could be adapted straightforwardly.

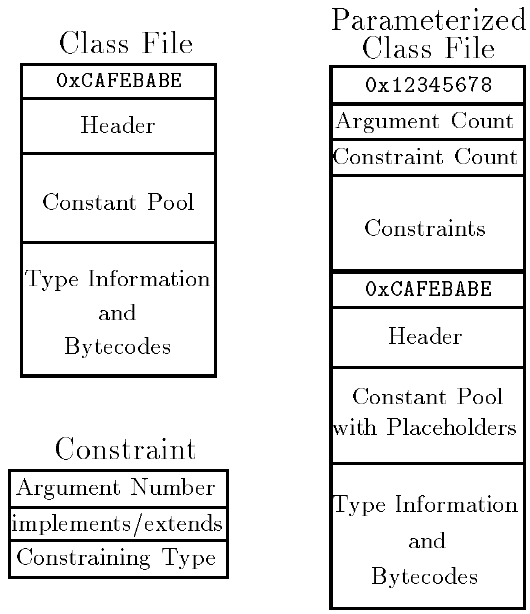

Figure 6: The original class file format, extensions for parameterized classes, and the format of a constraint.

```
class A {
   int i;
   void increment() {
     set(i + 1);
   }




   void set(int value) {
     i = value;
   }
 }


 A a;
 a.increment();
```

```
Method void increment()
  aload_0
  aload_0
  getfield #5 <Field A.i I>
  iconst_1
  iadd
  invokevirtual #6 <Method A.set(I)V>
  return

Method void set(int)
  ...




  aload_0    /* push a onto stack */
  invokevirtual #4 <Method A.increment()V>
```

Figure 7: Sample Java class and bytecodes

The entire binary representation for the parameterized class follows after the constraints. The only difference between this section and a standard class file is that special placeholders take the place of type names wherever parameters are used in the constant pool. In this paper, we will use `$1`, `$2`, etc. for these placeholders, but they can be any strings not appearing in the constant pool entries containing type information in a standard class file.

As discussed earlier, the Java Virtual Machine loads and links classes dynamically. The operations performed when a `ClassLoader` loads a class (or an interface) are:

- Fetch the bytes in the binary representation of the required class, perhaps from a file or over the network.
- Pass the bytes into the virtual machine to have them verified and turned into the virtual machine's internal representation of classes. This step produces a `Class` object.

All loaded classes are stored in a table so that they can be used again without reloading.

The design of the loader provides a great deal of flexibility in the way the binary representation of a class is located. We take advantage of this flexibility by augmenting the existing Java loader to create instantiations of parameterized classes. The steps involved in this process are listed below. Note that non-parameterized classes continue to be loaded in the usual manner. The steps to load an instantiation are:

- Convert the name of the class into a parameterized class name and argument names.
- Translate the argument names to class and interface types. This may involve loading other class files.
- Find the file containing the binary representation of the parameterized class and read it into memory.
- Check the parameter constraints against the arguments. If a constraint is violated, raise an exception.
- If the instantiation is valid, generate the binary representation of the instantiation from the parameterized class file and arguments; proceed in the usual manner to create a `Class` object for it. The bytecode verification process will ensure that the few remaining type checks described in Section 4.1 are performed.

Since the Java Virtual Machine specification does not allow class names of the syntactic form `Hashtable<Name, Integer>`, the compiler and `ClassLoader` must rewrite parameterized class names into a valid form. (This is similar to the way that inner class names are mapped onto the flat name-space of the virtual machine [Jav96]).

The heart of the implementation, when instantiating a parameterized class with a class or interface type, is the algorithm for transforming a parameterized class file into binary representations for different instantiations. For each parameter, we replace each occurrence of its placeholder with the actual argument's name in the constant pool. Figure 8 shows an example of this replacement method.

Once the replacements are made, the binary representation for the instantiation can be treated by the loader, verifier, and interpreter just as if it were an unparameterized class.

5.3 Instantiation with Primitive Types

The implementation technique described above does not work for instantiating classes with primitive types since different bytecodes are used for different primitive types. For example, the bytecode instruction for integer addition is different from the instruction for real addition. A load-time preprocessor can support instantiation with primitive types by modifying the bytecode instructions for parameterized classes. However, this is clearly a more complex transformation than required for class or interface arguments.

5.4 Discussion

By delaying instantiation until load time, we are able to achieve many of the stated design goals. Since the bytecode verifier and interpreter remain unaffected, type correctness and other properties of Java program execution are not compromised.

Our class file format for parameterized classes allows for separate compilation of a parameterized class and programs which create instances of it. Once the parameterized class file has been generated, a check against the constraints stored in that file guarantees type correctness of an instantiation, except for a few situations in which there is insufficient information in the constraints to guarantee this. As mentioned in Section 4.1, these cases can be checked by the compiler, but they must also be checked at load time since the Java Virtual Machine cannot assume all programs originate from a trusted compiler. We rely on the existing run-time bytecode verification process to catch these instantiations that conform to the declared constraints but are not valid classes.

```
Hashtable<$1,$2>.insert method:
   Method void insert($1,$2)
    aload_1
    invokevirtual #6 <Method $1.hashCode()I>
    istore_3
    aload_0
    iload_3
    aload_1
    aload_2
    invokevirtual #7 <Method Hashtable<$1,$2>.insertAt(IL$1;L$2;)V>
    return

Hashtable<Name,Integer>.insert method:
   Method void insert(Name,Integer)
    aload_1
    invokevirtual #6 <Method Name.hashCode()I>
    istore_3
    aload_0
    iload_3
    aload_1
    aload_2
    invokevirtual #7 <Method Hashtable<Name,Integer>.insertAt(ILName;LInteger;)V>
    return
```

Figure 8: Java bytecodes for the `insert` method as it appears in the `Hashtable` parameterized class file, and the same method instantiated as `Hashtable<Name, Integer>.insert`. The italicized names would be replaced by "mangled" names conforming to the rules for Java class names.

Load-time instantiation provides size and speed tradeoffs that are consistent with the rest of the Java system. The existence of only a single compiled class file for all instantiations of a parameterized class makes binary representations compact on disk and furthermore allow efficient distribution of Java classes over a network. At run time, there is some memory consumption increase, compared with a homogeneous implementation, because a new class is generated for each instantiation. Most Java Virtual Machines in use today do not share data structures between similar classes (such as the classes resulting from multiple instantiations of a parameterized class), leading to a memory demand increase proportional to the number of instantiations of the parameterized class. We have no absolute numbers yet, and cannot obtain such numbers until we have a better understanding of the extent to which parameterized types will be used in typical Java applications, but we expect that the memory increase will be small relative to the memory requirements for the system as a whole. This expectation seems to be confirmed by other systems, such as C++ and Ada, that use heterogeneous implementations of parameterization mechanisms.

Compared with a homogeneous implementation, load-time instantiation of parameterized classes increases the time spent loading classes during program execution, but in exchange enables faster execution because the resulting classes often contain fewer run-time type checks. Based on our preliminary tests and analysis of Java programs, we expect both the slow-down and the speed-up effects to be relatively small and have little impact on overall performance. Most Java applications today spend little time loading classes and performing run-time type checks.

It is important that the bytecodes produced for a parameterized class instantiation match the intended semantics of that language feature. For example, one would expect that adding an `Integer` object to the `Stack<String>` object in Figure 1 would be a type error, meaning that the bytecode verifier and interpreter should not allow that operation at run time. If it were allowed, a potential security problem would exist. Since parameterized class files are instantiated with the actual type parameters, our implementation prevents incorrect operations like this. However, implementations in which type variables are represented by a more general type than specified by the instantiation declaration may in fact compromise the language semantics in this way. This is true of some of the homogeneous implementations discussed in Section 6.

The idea of using a load-time preprocessor to expand parameterized classes is a general implementation tech-

Proposal	Language Extension				Implementation			
	Type Parameters			Inheritance	Instantiations use Shared Code		Instantiations use Separate Code	
	Constraints		No Constraints					
	implements/ extends	new type relation			Shared type info	Separate type info	Compile-time	Load-time
Pizza	X				X			
where Clauses		X				X		
C++			X				X	
Virtual Types				X			X	
This Paper	X							X

Figure 9: A summary of the language extensions and implementation techniques described in the different proposals. C++ is included as a reference point.

nique. It can be used to implement other proposals for type parameterization, as long as their semantics can be realized by a heterogeneous implementation. The preprocessor could also be used to experiment with other language features in the context of Java systems.

We have found the Java bytecode format to be an attractive representation when developing systems that use load-time expansion. The Java bytecode format offers architecture independence and a much higher abstraction level compared to more traditional systems, such as the C++ compile-time or link-time template instantiation that operates on machine code. The level of abstraction is particularly important when it may be necessary to report errors at instantiation time. Without high-level format and type information, it can be difficult to detect and report errors in a meaningful way (i.e., relate them to the source code).

5.5 A Prototype Implementation

A prototype of the virtual machine extension described in Section 5.2 has been implemented and tested using Sun's JDK 1.0.2 source release. While a full compiler for parameterized classes has not been implemented, this prototype is sufficient to test the effectiveness of load-time instantiation. The implementation consists of approximately 500 lines of Java code and a ten line change to the default loader in the Java Virtual Machine. A successful prototype was built as a subclass of the `ClassLoader` class, but we found that this was too restrictive for general use since we could not install that `ClassLoader` as the default for the Java Virtual Machine. Therefore, we made one minor change to the underlying default loader to make it recognize parameterized classes. Although this prototype makes several minor assumptions (such as "_" not appearing in any class name and that all classes are loaded by the same `ClassLoader`), it is clear that the changes to Java's runtime environment are very straightforward and isolated to one small section of the virtual machine. In a later implementation, we plan to remove the limitations on the existing version and to add primitive type arguments, which were also left out of the original prototype.

In addition to creating parameterized class files for all examples in this paper, we have taken several classes from the `java.util` package, such as `java.util.Hashtable`, and transformed them into parameterized classes. Making these new classes required virtually no change to the existing code other than replacing `Object` with the parameter identifiers, `$1`, `$2`, etc., and inserting constraint information at the top of the class files. The run-time speed of the parameterized versions of these classes improved slightly. To cite one specific example, a program which populated a `java.util.Hashtable` and then performed one million lookup operations increased in speed by 2–3% when the hash table was replaced by our parameterized version of it.

6 Related Work

There have been several other proposals for adding parameterized types or generic classes to Java. Although it may appear that the language extensions from these proposals are equally expressive and that the different implementation strategies are more or less equal in power, fundamental differences do exist between them. This section discusses the other proposed designs, touching on the distinguishing features, as well as potential advantages and disadvantages, of each. The language and implementation choices of all proposals described below are summarized in Figure 9.

6.1 `where` Clauses

One proposal, described in [MBL97], presents a Java extension similar to `where` clauses in CLU [LSAS77]. There are several semantic differences between that constraint mechanism and the ones used in both Pizza (discussed below) and this paper. On the positive side,

```
class BroadcastList<C extends Channel> {
  C channels[];
  void add(C c) { ... }
  void broadcast(String s)  {
    int i;
    for (i = 0; i < channels.length(); i++)
      channels[i].send(s);
  }
  ...
}

class EncryptedChannel extends Channel;
class UnencryptedChannel extends Channel;

BroadcastList<EncryptedChannel> list = ...;
list.add(new EncryptedChannel());
(*)
list.broadcast("Hello");
```

Figure 10: A list of `EncryptedChannels` that could be attacked by handwritten bytecodes to add an `UnencryptedChannel` if `C` does not contain the exact type of the argument for each instantiation.

`where` clauses allow the programmer to state that a class conforms to a certain constraint without explicitly declaring the relationship when the class is defined. In our `Hashtable` example, `where` clauses would allow the programmer to use any class that has a `hashCode` method as the `Key` instead of just those declared to implement the `Hashable` interface. If only existing Java constructs for subtype relations are used, namely `implements` and `extends`, this type of constraint cannot be expressed. However, since this problem in Java arises in situations outside of parameterized classes, it seems more reasonable to solve this problem for Java as a whole instead of introducing a new type relation only in the context of parameterized class constraints. Section 7 describes two possible Java extensions that allow subtyping relations to be declared without changing class declarations.

The main implementation described in [MBL97] for `where` clauses changes the bytecodes and virtual machine significantly in order to allow shared code, but separate constant pool information, among instantiations of a parameterized class. This makes the implementation more costly and the virtual machine more complicated than our method. Any change to the bytecodes and bytecode verifier will require all safety and security aspects of the system to be reevaluated. Clearly, this could be an undesirable and daunting task. An implementation based on load-time instantiation would be possible for `where` clauses by adding more forms of constraints to those described in Section 5.

6.2 Pizza

Pizza [OW97b] is another parametric class extension using a constraint mechanism based on F-bounded quantification [CCH+89] and the `extends` and `implements` type relations. However, the type parameters in Pizza are prevented from appearing in certain places were class names may appear. For example, two errors occur in the following program when compiled in Pizza [OW97a]:

```
class List<A> {
  static A a;    // not allowed in Pizza
  void newA() {
    A a;
    a = new A();  // not allowed in Pizza
  }
}
```

Both statements would be allowed in our implementation and the `where` clause approach of [MBL97]. Another restriction is that a class cannot inherit from a parameter, making mix-ins impossible to express.

Although these limitations may result from a different model of parametric types, the homogeneous implementation used by the Pizza compiler makes these features difficult, if not impossible, to implement. Pizza creates a single class file for each parameterized class. This class represents each type argument as the most general possible type for that argument. The compiler then inserts casts into the bytecodes that use instances of a parameterized class to ensure type correctness. These casts are similar to those discussed in the context of the `Stack` class and other examples in Section 3, but are added by the compiler instead of by the programmer. This strategy does allow the Java Virtual Machine to remain unchanged, but it clearly limits how type variables can be used and also still requires run-time casts. Our implementation does not have these limitations since the instantiation of a parameterized class replaces type variables with the actual type of an implemented class, allowing type variables to appear wherever a class name may appear.

It is also worth keeping the security implications of each implementation technique in mind. If the type used in the place of a parameter is more general than the actual argument type in an instantiation, as it is in Pizza, there is the possibility that hand-coded bytecode could take advantage of this discrepancy. Specifically, compiler type checking and the compiler-inserted run-time checks guaranteeing type safety when parameterized types are used could be circumvented by using bytecodes not generated by the Pizza compiler.

For example, someone may be able to insert an unencrypted channel into a list of encrypted channels used by a library for secure communication, as shown in Figure 10. In Pizza, the `EncryptedChannel` instantiation of `BroadcastList` would use bytecodes based on `C` having the type `Channel`, meaning `BroadcastList<EncryptedChannel>.add` would take an argument of type `Channel`. If we manually inserted bytecodes to add an `UnencryptedChannel` object to `list` at position `(*)`, there would be no way to catch this error in the bytecode verifier or at run time since an `UnencryptedChannel` is a subtype of `Channel`. The only way to prevent this is to give `BroadcastList<EncryptedChannel>.add` a type where the parameter can only be an `EncryptedChannel`.

As discussed in [DFW96], it is very important that the semantics of the Java language matches the semantics of the generated bytecode instructions. In that paper, the authors discuss a situation where valid bytecodes are able to perform an action prohibited by the language. A similar situation occurs in Figure 10. Bytecodes that are seemingly valid to the Java Virtual Machine do in fact perform an operation which the language restricts. In order to maintain the desired security properties, such situations must be prevented. An alternative heterogeneous implementation based on C++ -style macro expansion, also discussed in the Pizza paper, does eliminate this problem, but it is unclear whether that implementation would be able to support separate compilation of parameterized classes without changing the class file in a way similar to our method.

6.3 Virtual Types

Virtual types have also been suggested as a way to implement parametric types [Tho97]. The main idea is to allow type members of classes that are "virtual" in the sense of C++ virtual members, with derived classes allowed to redefine a virtual type member to any of its subtypes. This idea can be used in a manner similar to parameterized types, since in each subclass, the virtual type could be given the same value as the argument to the corresponding parameterized class. In general, redefining a type member in a subclass does not create a subtype due to contravariance. To ensure that a subtype is created, the virtual types proposal forces covariance through run-time casts and type checks. This is similar to the checking required to ensure type safety of Java's covariant arrays. In addition to the loss of static typing for parameterized classes due to these required casts, tying genericity to inheritance also limits expressiveness. For example, mix-ins can not be expressed.

In the implementation described in [Tho97], compiler generated virtual methods and casts perform run-time type checks and typecasts needed to ensure type correctness of all contravariant uses of virtual types. Due to this reliance on compiler generated code, the system is potentially susceptible to the same type of security loopholes as described above.

6.4 Run-time Efficiency

One final point of comparison between these methods is that inserted type casts add run-time overhead. The cost of casts used for type correctness of instantiations have been measured to be anywhere from 1% to 5% [Tho97] of the run time, with degenerate cases costing up to 17% [MBL97]. Our own measurements are within the range cited in [Tho97]. Considering Java's design goals, this penalty is not overly restrictive, but it should still be considered in the tradeoffs between different proposals. This overhead may become more or less significant as better optimized bytecode and native compilers are used for Java programs.

7 Additional Language Issues

There are two main avenues for future work. The first is to examine some remaining language issues. A second, more tentative, future direction involves studying aspects of the Java language that have been more problematic for us and suggesting language modifications.

Two language design issues that require further examination are the exact format of `extends` and `implements` clauses and interactions between overloaded methods and parameterized classes. Overloaded

```
class B extends A { ... }

class Printer {
  void print(A a) { ... }
  void print(B b) { ... }
}

class C<T extends A> {
  void callPrint(Printer p) {
    T temp;
    ...
    p.print(temp);
  }
}

C<B> c = new C<B>();
Printer printer = new Printer();
c.callPrint(printer);
```

Figure 11: An example of overloaded methods and parameterized classes.

methods could either be resolved on a per-instantiation basis, or once for all instantiations. Figure 11 shows a situation where the two resolution strategies result in different method invocations.

We also believe it will be useful to examine some of the Java type restrictions and determine whether they can be made more lenient. Two specific items of interest are specification of constructors and post-facto declaration of subtype relationships.

Constructors are not inherited by subclasses and cannot be specified in an interface. As a consequence, there is no way to specify that a type variable will always be instantiated with a class that has a certain constructor available. Thus, requiring the presence of a constructor falls into the category of type rules which must be checked on each instantiation individually. However, it may be beneficial to be able to express the presence of constructors in parameter constraints. The `where` clause proposal described in Section 6.1 allows explicit constraints on the presence of constructors, and that aspect of `where` clauses could be adapted to our method. A more suitable solution may be to allow constructors to be specified in an interface, but it remains to be seen if this is a feasible addition to the Java language.

The inability to specify a subtype relationship for an existing class limits not only the expressive power of our constraint mechanism but the expressive power of the Java language as a whole. One way to alleviate this limitation would be through an `implementedBy` declaration. An example is shown in Figure 12. Structural conformance has also been proposed as a solution to this problem [LBR96]. Any class implementing all the methods listed in an interface could be used as a value of that interface type, regardless of whether the class was declared to implement it or not. There are many issues and tradeoffs involved with structural conformance, such as accidental conformance, and it is unclear whether or not it is a good match for the Java language.

8 Conclusions

This paper describes a parameterized type mechanism for the Java programming language and an implementation based on inserting a preprocess step into the load phase of the Java Virtual Machine. The language extension, based in part on an analysis of parameterization in object-oriented languages reported in [CCH+89] and essentially similar to the extension considered in [OW97b], appears adequate for most common situations. It uses only type relations `implements` and `extends`, which are familiar to Java programmers.

In comparison with competing proposals described in Section 6, our implementation is relatively simple and supports more flexible language extensions than others. By delaying instantiation until load time, we are able to achieve what we believe is an appropriate balance between language expressiveness, run-time efficiency and compiled code size. Since the changes to the Java virtual machine are restricted to the loader, we expect that our implementation can be easily added to most Java virtual machines. Moreover, use of the standard verifier and bytecode interpreter means that our Java language extensions do not compromise any security properties. We believe that this language extension will be easy for Java programmers to adopt and use effectively, and interfere minimally with any current

```
class Name {
  int hashCode() { ... }
  ...
}

interface Hashable implementedBy Name {
  int hashCode();
}

class Hashtable<Key implements Hashable, Value> { ... }

Hashtable<Name, Integer> table = new Hashtable<Name, Integer>();
```

Figure 12: Specifying an `implements` relationship after a class has been declared.

or future developments in Java implementation.

Our prototype implementation of an extended Java Virtual Machine demonstrates that load-time instantiation is an effective mechanism with many clear benefits. Although we believe that language extensions should be considered slowly and conservatively, it is possible that our implementation strategy using the Java loader may be a useful starting point for other potential extensions to the Java language.

Acknowledgments: We would like to thank Guy Steele for his comments and suggestions, including the `implementedBy` construct shown in Figure 12. We also thank Gilad Bracha, David Detlefs, and Kathleen Fisher for their many useful discussions and comments.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to netscape and beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 190–200, 1996.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Jav96] JavaSoft. Innerclasses in Java 1.1. available from http://java.sun.com, 1996.

[KLM94] D. Katiyar, D. Luckham, and J.C. Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[L+81] B. Liskov et al. *CLU Reference Manual.* Springer LNCS 114, Berlin, 1981.

[LBR96] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance in java. Technical Report CSD-TR-96-077, Purdue University, December 1996.

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

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[Mil85] R. Milner. The Standard ML core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[MS95] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1995.

[OW97a] Martin Odersky and Philip Wadler. The Pizza compiler. available from http://www.math.luc.edu/pizza, February 1997.

[OW97b] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[PSLM96] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The Standard Template Library.* Prentice-Hall, 1996.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference On Object Oriented Programming*, 1997.

[Ull94] J.D. Ullman. *Elements of ML programming.* Prentice Hall, 1994.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.