Parameterized Types in Java

Andrew C. Myers Joseph A. Bank

{andru, jbank}@lcs.mit.edu
MIT Laboratory for Computer Science

1 Introduction

Java [8] promises to be an important language for WWW applications and software development in general. This popularity arises from several factors: simple object-oriented design; syntactic similarity to C++ [7]; strongly-typed code that is statically verifiable both as source code and as intermediate code.

This note describes how to make Java more expressive by adding parametric polymorphism, which increases the power of compile-time type analysis and ultimately makes Java programs faster. We do not compromise the safety properties of Java, and show how simple extensions to the Java bytecodes allow efficient verification of parameterized intermediate code. Finally, performance results from an extended Java interpreter that we have written suggest that these extensions support high-performance implementations. Further details on our extensions are available [1].

2 Java Limitations

Java's support for generic code (e.g. generic collections) is limited to subtype polymorphism. For example, the hash table class is declared to map from Object → Object, where Object is the root of the class hierarchy. Hash table users must rely on run-time type casting to extract values from the hash table. To cast these values safely, run-time type checking is required, and its relative cost will probably increase as execution of Java becomes more sophisticated (i.e., on-the-fly compilation). As we will see, parametric polymorphism permits the declaration of a hash table that maps between specific types, eliminating run-time checks.

3 Design Overview

The design sketched here for parametric polymorphism in Java is derived from that in Theta [5] and has its ancestry in CLU [4]. However, Java has the additional constraint that it must be compilable into easily verifiable bytecodes. We have also designed a parameterized bytecode specification that preserves efficient verifiability.

The extensions to Java for parametric polymorphism are based on constraining generic code with *where clauses*. Because generic code is constrained, uses of generic abstractions can be type checked without examining the implementations of their abstractions. Many languages have macro-like generics (e.g., C++ and Modula-3 [6]) that do not provide this kind of separate compilation.

Where clauses are a more powerful constraint mechanism than subtypes [3] because they can safely express the requirement for binary methods such as equal. They are also more powerful than the usual formulations of F-bounded polymorphism [2].

Our extended bytecode specification is probably the only example of a statically typed instruction set that supports parametric polymorphism. Unlike many generic-code implementations (e.g., for C++ templates), one sequence of instructions suffices for all instantiations of the generic code. In addition, the bytecodes can be efficiently verified in their generic form, which is sufficient to prove all instantiations correct.

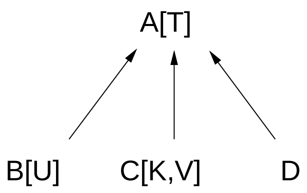
4 Language Extensions

Java contains interface and class definitions. We allow both kinds of definitions to be parameterized over any number of parameter types. For example, a parameterized map class might appear as follows:

```
class Map [Key, Value] extends Set[Key]
  where Key { boolean equal (Key k); } { ... }
```

Within the body of the class (indicated by ellipses), the formal identifiers Key and Value may be used as types. In the implementation of Map, the only operation that may be performed on values of type Key is the

Andrew Myers is advised by Barbara Liskov and is supported in part by ARPA contract N00014-91-J-4136. Joseph Bank is supported by a grant from Sun Microsystems. Web: www.pmg.lcs.mit.edu



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

Figure 1: A parameterized class hierarchy

where-clause operation equal; no operations may be performed on objects of Value. This restriction is necessary for separate compilation. The clause extends Set[Key] indicates that for all Key and Value types, Map[Key, Value] inherits from Set[Key].

A parameterized interface or class is *instantiated* by providing actual types for the parameter types; thus, Map[int,Set[string]] represents a map from integers to sets of strings. An instantiation is legal when the actual types *satisfy* the where clauses of the parameterized abstraction. Where clauses may also mention static methods, providing additional expressive power.

5 Verifying Bytecodes

Java bytecodes are a statically-typed instruction set. To support parametric polymorphism, we added an instruction to invoke the operations described by where clauses. Minor changes were also made to allow the description of instantiation types.

The changes to the bytecodes still permit quick verification. The Java bytecode verifier currently uses standard dataflow analysis techniques; it soundly infers the types of the contents of registers and stack locations, using static type information that is contained in some bytecodes such as method invocation.

The core of the verifier constructs the union of two types that describe values which may be placed in a storage location at the same point in the execution. This union is conservatively approximated by the least supertype of the two types in the type hierarchy. To support parameterization, the merge procedure is modified. When finding the least supertype, extends clauses are used to rewrite parameter names along the chain. Consider the class hierarchy shown in Figure 1 with its corresponding extends clauses. The union of $B[X]$ and $C[X, Y]$ is approximated as follows:

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

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

Figure 2: Collection Class Run Times

6 Results

We modified the Sun implementation of the Java Virtual Machine to support our extended bytecodes. Our goal was to provide good performance without duplicating executed code. The interpreter extensions slowed execution of non-parameterized code by only 4%; a careful implementation would have no slowdown [3]. We also tested the performance of a collection class implemented with parameterization and with other techniques. The results are shown in Figure 2: the 17% speedup offered by parameterized classes would likely be larger if Java were compiled, since run-time type casting of Object would be relatively more expensive. The results also show that the performance is penalized only 4% for using generic code rather than hard-wired element types.

References

[1] Joseph Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. At `ftp://ftp.pmg.lcs.mit.edu/pub/thor/javapt.ps.gz`, 1996.

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

[3] 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. At `ftp://ftp.pmg.lcs.mit.edu/pub/thor/where-clauses.ps.gz`.

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

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

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

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

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