Overview of PolyJ

Barbara Liskov **Nick Mathewson** **Andrew Myers**

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

1 Introduction

PolyJ is an extension to Java that supports constrained parametric polymorphism. It allows abstractions to be parameterized so a single piece of code can implement many related abstractions. For example, PolyJ can be used to define a sort routine that sorts arrays of ints, arrays of floats, etc. It can also be used to define homogeneous collections, e.g., a parameterized Set type where one set object has ints as elements, another has Strings as elements, etc.

Any class or interface declaration can be parameterized. A parameterized declaration must be *instantiated* to be used, by providing an actual type for each parameter. For example, if Set is a parameterized class with a single type parameter, then Set[int], Set[Object], Set[char[ ]], and Set[Set[int]] are examples of instantiations. Note that arrays, and primitive types like int, can be used as parameters.

The result of an instantiation can be used as a first-class type. For example, each instantiation of Set can be used as a type, so that variables of that type can be declared, objects of the type can be created, and methods of those objects can be called. An instantiation may not, however, be used as an exception type.

In general, not all types make sense as actual parameters; for example, a sort routine needs some way to determine whether one element is less than another. A parameterized declaration can limit what types can be used as parameters by stating *constraints* on its parameters. Constraints identify methods that a type provided in an instantiation must have.

PolyJ's mechanisms are similar to C++ templates, but they provide better support for separate compilation. The PolyJ compiler does complete type checking of parameterized declarations and instantiations, even when they are compiled separately. The code of parameterized declarations can use only the parameter methods indicated by their constraints; instantiations are required to supply the methods indicated by the constraints. Otherwise there will be a compile-time error. This checking ensures that even though a parameterized declaration is compiled separately from code that uses it, there will be no type errors at run time. In particular, a method assumed to exist for a parameter in a parameterized declaration will actually be present in any instantiation, with an appropriate signature.

The PolyJ compiler produces Java classes and then calls a Java compiler. The implementation is much more space-efficient than the one for C++ templates. In C++, there is no sharing of code between different instantiations of the same template. However, in PolyJ almost all code is shared among the different instantiations of the same parameterized declaration.

In addition to the compiler, this release includes a number of parameterized collection types in the polyj.util package. This package contains parameterized versions of all the collections defined in java.util for Version 1.0 of the JDK [GJS96].

2 Headers of Parameterized Declarations

A parameterized declaration defines a group of related classes or interfaces, one for each legal instantiation. Any class or interface declaration can be parameterized except for exception classes, i.e., classes that are subclasses of Exception or one of its subclasses. For example, the following code is a parameterized interface, Set.

```
interface Set[T] where T { boolean equals(T);
                           int hashCode(); }
{
   boolean add(T elem);
   T remove(T elem) throws NoSuchElementException;
   boolean contains(T elem);
   ...
   Enumeration[T] elements();
}
```

The interface Set has a single type parameter, T. This type is required to have hashCode and equals methods, which all object types are guaranteed to provide. Note that the type T is used within the declaration of Set, and is even used to instantiate another parameterized interface, Enumeration.

As in this declaration, the header of any parameterized declaration indicates what instantiations are legal by specifying parameters and constraints on each parameter. A class declaration has the following form (following the conventions in Section 2.4 of the Java language specification [GJS96]):

ClassDeclaration :
    Modifiers$_{opt}$ class Identifier Params$_{opt}$ Constraints$_{opt}$
        Super$_{opt}$ Interfaces$_{opt}$ ClassBody

The form of a parameterized interface is similar:

InterfaceDeclaration :
    Modifiers$_{opt}$ interface Identifier Params$_{opt}$ Constraints$_{opt}$
        ExtendsInterfaces$_{opt}$ InterfaceBody

The optional *Params* lists the names of one or more parameters:

Params :
    [ParameterList]
ParameterList :
    Identifier
    ParameterList , Identifier

Each of these parameters names a type. The optional *Constraints* gives names and signatures of methods that the parameter types must have:

Constraints :
    Constraint
    Constraints Constraint
Constraint :
    where Identifier { WhereDeclList }
WhereDeclList :
    WhereMethodHeader ;
    WhereDeclList WhereMethodHeader ;

The *WhereMethodHeader* is the same as a Java method header except that the only modifier allowed is static.

Public methods of Object (e.g., toString, hashCode) need not be listed in constraints, with one exception: if the equals method is required, the parameterized declaration must provide a constraint to this effect. Usually this constraint will specify an argument of the parameter type, so for a parameter T, the constraint will be

```
boolean equals (T);
```

Note that Object's equals will satisfy this constraint, but a more specific notion of equality for the actual type will be preferred.

For example,

```
class BoundedSet[E] where E { boolean equals(E); } {
  public BoundedSet( ) { ... }
  public insert(E element) { ... }
  public E pickElement( ) throws EmptyException { ... }
}
```

states that the class BoundedSet has one parameter, E, and this type must have an equals method.

As a second example, consider

```
class Map[K,V] where K { boolean equals(K key); byte getHashByte( ); }
          where V { boolean equals(V value); }
          extends Collection[K]
          implements Cloneable {
  ...
}
```

Here, Map is defined as taking two parameters, each of which has an associated where clause. In this case, the definer has chosen to provide names for the arguments of the required methods; providing argument names in where clauses is optional, and any argument names provided are ignored.

BoundedSet and Map are both parameterized types. It is also possible to use parameterized classes to define groups of procedures (static methods), e.g.,

```
class CollectionProcs[C, T]
    where C { int size( ); T elementAt(int); void setElementAt(T, int); }
```

```
    T { boolean less (T); } {

  public static sort ( C coll ) { ... }
  ...
}
```

The methods in this class handle all collections that can be accessed using the elementAt, setElementAt, and size methods, provided their elements have a partial order. Note that a constraint for a particular parameter can use the other parameters in the method signatures, e.g., the constraint for parameter C makes use of T.

Where clauses can list exceptions that methods can throw, e.g.,

```
class Bar[K] where K { K smaller ( ) throws LeastException; } ...
```

Checked exceptions *must* be listed (or instantiations will not be legal; see Section 3). Unchecked exceptions need not be listed, as in the CollectionProcs example above.

The above where clauses place constraints on methods of the type's objects. It is also possible to constrain parameters to have constructors and static methods, e.g.,

```
class Foo[N] where N { N(int); // a constructor
      static boolean compare(N, N); // a static method
      boolean less(N); // an object method
  ...
}
```

3 Instantiation

Parameterized classes and interfaces must be *instantiated* before they can be used. This is accomplished by providing actual types in place of the formals. The instantiation will succeed *only* if the actual types satisfy the constraints stated in the where clauses, i.e., they have the methods listed for the corresponding parameters in the where clauses.

An instantiation has the form:

```
Instantiation :
        Identifier [ TypeList ]
TypeList :
        Type
        TypeList, Type
Type :
        PrimitiveType
        ReferenceType
        ParameterType
        Instantiation
```

The identifier in an instantiation must name a parameterized class or interface. *PrimitiveType* is a built-in type (such as int); *ReferenceType* includes both array types and types defined by non-parameterized classes and interfaces; and *ParameterType* is a parameter of a parameterized declaration. The use of primitive and array types in instantiations is discussed further in Section 4.

An instantiation has members whose types are derived from the parameterized declaration in a straightforward way: the actual type replaces the corresponding formal in all method signatures and member variable declarations. Thus BoundedSet[int] has methods:

```
BoundedSet[int]( ); // the constructor
void insert(int); // to add an element to the set
int pickElement( ) throws EmptyException: // to get an element of the set
```

An instantiation can be used in all appropriate ways. If the parameterized module defines a type, the instantiation can be used wherever a type can be used; if the parameterized module provides only static members, the instantiation can be used to access these members.

For example, BoundedSet[int] can be used like a regular type, e.g.,

```
BoundedSet[int] s = new BoundedSet[int]( );
s.insert(5);
```

Similarly, ArrayProcs[char] can be used to sort arrays of characters:

```
char [ ] c = new char [100] ( );
... // initialize c
ArrayProcs[char].sort(c);
```

An instantiation is legal only if the corresponding actual types satisfy the constraints listed for those type. Otherwise there will be a compile-time error. First the constraint is rewritten by replacing type parameters with the corresponding actuals. Then the compiler attempts to match the constraint to a unique method of the type. Matching is done exactly as in Java to determine what method to call; here the constraint plays the role of the method invocation. (See Section 15.11.2 of the Java language specification [GJS96].) If there is more than one method of the same name and with compatible argument types, the closest match is selected; if there is no closest match, there is a compile-time error. Furthermore, if the selected method's result type isn't a subtype of the constraint's result type, or if it has a checked exception that isn't a subtype of an exception listed in the constraint's exceptions, there is a compile-time error. If there is no error, the selected method is what will actually be called by the parameterized declaration for that instantiation.

For example, consider the instantiation BoundedSet[Foo], where Foo has two equals methods:

```
boolean equals (Object); // method inherited from Object
boolean equals (Foo); // method defined for Foo
```

Both of these declarations satisfy the constraint

```
where Foo { boolean equals (Foo); }
```

The one selected is the one that takes the Foo argument, since it is the closest match; this is the same selection that would be made for a call, e.g., f.equals(g), where f and g are Foos. The selection is legal because its result type is what the constraint specifies, and because it does not throw any checked exceptions. And when code for BoundedSet[Foo] runs, it will call this selected method when it uses the parameter's equals method.

4 Support for Primitive Types and Arrays

PolyJ allows parameterized classes and interfaces to be instantiated with Java primitive types and Java array types. Java defines no methods for primitive types and very few methods for arrays, which limits their ability to satisfy constraints. Therefore, PolyJ provides them with a number of methods. The additional methods cannot be invoked directly using a standard method call; instead, they can be used only to satisfy constraints.

For example, the following code is legal:

```
class Summation[NumT] where NumT { NumT add(NumT); } {

  public NumT val;

  public Summation(NumT initialValue) { val = initialValue; }
  public void addValue(NumT value) { val = val.add(value); }

} // end Summation

public class Demo {

  public static void main (String[ ] args) {
    Summation[int] SI = new Summation[int](3);
    SI.addValue(4);
    SI.addValue(3);
    System.out.println(SI.val);
  }

} // end Demo
```

This code is legal because int has an add method for the purpose of satisfying constraints, even though int does not have any methods in Java.

The Appendix describes the methods PolyJ provides for arrays and primitive types.

5 Parameterized Declarations

A parameterized declaration can use its parameters as types in declarations and instantiations, with one exception: a parameterized declaration *cannot* implement or extend one of its parameters. For example

```
class Foo[T] extends T { ... }
```

is illegal. (However, the parameterized declaration can be a subtype of an instantiation, as discussed in Section 7.)

Furthermore, the parameterized declaration can use methods of its parameter types that where clauses require them to have. Each instantiation is guaranteed to provide, for a parameter, all the methods listed in the constraints for that parameter, plus the public Object methods (except for equals). Therefore the parameterized declaration can use these methods. However, it cannot use any other methods or there will

```
public class BoundedSet [E] where E { boolean equals(E); } {
  private E[ ] els;
  private int sz;

  public BoundedSet ( ) { els = new E[100]( ); sz = 0; }

  public void insert ( E x ) throws NullPointerException, FullException {
    if (x == null) throw new NullPointerException( );
    for (int i = 0; i < sz; i++) if (els[i].equals(x)) return;
    if (sz < 100) { els[sz] = x; sz++; }
    else throw new FullException( );
  }

  public void delete (E x) {
    for (int i = 0; i < sz; i++)
      if (els[i].equals(x)) {
         sz --;
         els[i] = els[sz];
         els[sz] = E.default;
         return;
      }
  }

  public String toString ( ) {
  // uses toString of E to produce a string representation for the set
  }
}
```

Figure 1: A partial implementation of BoundedSet.

be a compile-time error. Also, each method use must be type correct. For methods specified in constraints, the signature in the constraint is used for type checking.

Code in a parameterized declaration can use a special form, T.default, to denote the default value for that type. In instantiations where T is bound to a non-primitive type, T.default is null; for instantiations where T is bound to a primitive type, T.default is the default value for that type, e.g., 0 for int. It is not legal to assign null to a T variable since null does not make sense for instantiations where T is bound to a primitive type; however, T.default can be assigned. It is legal to compare the content of a T variable with null (using == or !=); this test will always fail in instantiations that bind T to a primitive type.

Figure 1 gives part of a simple implementation of BoundedSet, in which a particular set object can hold no more than 100 elements. The representation consists of a 100-element array plus an int to store the current size of the set. Note the form for the constructor. Within a parameterized class, constructors are simply named with the class name, without any parameters. (However, when the constructor is called, as shown previously, parameters must be supplied.)

Both insert and delete use the equals method of parameter E. This code will call the method selected for the instantiation for which it is being called. For example, consider:

```
BoundedSet[Foo] foos = new BoundedSet[Foo]( );
foos.insert("a");
```

When the call foos.insert runs, it will call the Foo equals method with signature

```
boolean equals (Foo);
```

BoundedSet also uses the toString method (in its own toString method); toString need not be mentioned in the constraints, however, because it is a public method of Object.

The code illustrates the use of null and the special default value. The insert method rejects null as an input to ensure that all members of a set are actual objects. The comparison with null can only succeed, however, in instantiations, such as BoundedSet[String], in which the parameter E is bound to a non-primitive type. The implementation of delete uses the default value to clear the array element that is no longer being used, so that the object it used to point to can be garbage collected.

If a parameterized class has any class or instance variables of a parameter type, they will be initialized by Java to null; the code of the parameterized class must initialize those variables before they are used. Otherwise NullPointerException will be raised at such a use in an instantiation where that parameter is bound to a primitive type. For example, suppose BoundedSet had the following (erroneous) implementation:

```
public class BoundedSet[E] where E {boolean equals(E); } {
  private E recent;
  private E[ ] els;
  private int sz;

  public BoundedSet( ) { els = new E[100]( ); sz = 0; }
  E pickElement ( ) throws EmptyException { return recent; }
  ...
}
```

and consider

```
BoundedSet[int] b = new BoundedSet[int]( );
int x = b.pickElement( );
```

This call of pickElement will throw NullPointerException (since null does not represent any int).

A parameterized declaration cannot contain overloaded methods that would be ambiguous in some instantiations. Therefore the following is illegal:

```
class Map[K,V] ... {

  public void m(K x) { ... }
  public void m(V x) { ... }

  public void m2(K x) { ... }
  public void m2(int x) { ... }
}
```

The overloading of m is illegal because the two declarations would be ambiguous in any instantiation where the same type is bound to K and V, e.g., Map[String,String]. The overloading of m2 is illegal because the two declarations would be ambiguous in any instantiation where int is bound to K, e.g., Map[int,String].

6 Static Members of Instantiations

Static members of a parameterized class exist on a per-instantiation basis, rather than on a per-class basis. For example, Map[int,String].staticField is a different field from Map[int,int].staticField. Static variables are initialized the first time that particular instantiation is used in a program.

This semantics allows a parameterized class to have static variables whose type depends on a parameter. E.g., in Map we might have:

```
static K lastKey;
```

7 Subtype Relations among Instantiations

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

```
class BoundedSet[E] where E { boolean equals(E); }
  extends SimpleBoundedSet[E]
  { ... }
```

Such a subtype declaration is legal if the new subtype satisfies satisfy the constraints of the supertype. For example, the declaration for BoundedSet is legal if the constraints for SimpleBoundedSet either are empty, or require

```
boolean equals(E);
```

Otherwise the declaration is illegal and there will be a compile-time error. (Recall that PolyJ does not allow a parameterized declaration to be a subtype of one of its parameters.)

A legal subtype declaration like the one for BoundedSet states that for any actual type Foo that can be used to instantiate BoundedSet, BoundedSet[Foo] is a subtype of SimpleBoundedSet[Foo]. E.g., BoundedSet[String] is a subtype of SimpleBoundedSet[String], and BoundedSet[int] is a subtype of SimpleBoundedSet[int].

The supertype in the declaration need not have the same number of parameters as the subtype. For example, in the declaration of Map we had

```
class Map[K,V] ... extends Collection[K] implements Cloneable ...
```

Thus for all legal instantiations, Map[Foo,Bar] is a subtype of Collection[Foo] and Cloneable. Furthermore, all legal instantiations are subtypes of Object, e.g., Map[int,String] is a subtype of Object.

However, an instantiation is *never* a subtype of another instantiation that differs from it in one of the actual parameters. For example, BoundedSet[Foo] is not a subtype of BoundedSet[Bar], even when Foo is a subtype of Bar. Similarly, BoundedSet[Foo] is not a subtype of SimpleBoundedSet[Bar]. The reason for this restriction is easy to understand. Consider the following code:

```
BoundedSet[Bar] b = new BoundedSet[Foo]( );
b.insert(new Bar( ));
BoundedSet[Foo] f = (BoundedSet[Foo]) b;
Foo x = f.pickElement( );
```

If the subtype relationship were allowed, the initial assignment to b would be legal. Clearly the call of insert, with a Bar element, would also be legal, since the signature of insert for this instantiation is:

```
void insert (Bar);
```

Furthermore the cast (in the third statement) would succeed since b is a BoundedSet[Foo] object, and therefore the assignment to x would be executed. But this assignment would cause x to refer to a Bar object, which is not type correct.

PolyJ avoids this problem by ruling out the initial assignment to b, since BoundedSet[Foo] is not a subtype of BoundedSet[Bar]. As a result of this rule, we can be sure that a BoundedSet[Foo] object contains only elements that are Foos or some subtype of Foo. Therefore, it will never be possible for the call on a BoundedSet method such as pickElement to return an object whose type is not Foo or a subtype of Foo.

The rule used in PolyJ ensures that homogeneous collections contain the right type of element through the use of compile-time checking [DGLM95]. Java does not use this rule for for arrays, however; e.g., Foo[ ] is a subtype of Bar[ ] if Foo is a subtype of Bar. This means that it is legal at compile time to store a Bar element into a Foo[ ] object. To prevent a Foo[ ] object from containing Bar elements, Java does a runtime type check each time an element is stored into the array. PolyJ avoids the need to do the runtime check.

8 Current Limitations

Our current implementation has some limitations. The most significant one is that at present we support only Java 1.0, and therefore inner classes are not supported. In addition:

- It is not possible to run the main method of a parameterized class from the command line.
- A class can implement at most one instantiation of an interface. For example,

  ```
  class Foo implements Bar[Integer], Bar[String] {...}
  ```

  is not allowed, because methods of the same name from each interface cannot be distinguished in the translation.
- Methods inherited from a parameterized superclass do not satisfy a non-parameterized interface. For example, in

  ```
  class Foo[T] { void m(T x) {...} }
  interface Bar { void m(Integer x) {...} }
  class Baz extends Foo[Integer] implements Bar {...}
  ```

method m of Foo[Integer] will not be recognized as the method required by the Bar interface. However, a wrapper method may be written to achieve this.

- Within a parameterized class, OP= operators (such as +=, *=) cannot be used on static fields. For example,

```
class C[T] ... {
  static int x;
  static int y;
  ...
  boolean aMethod( ) {
    x += 6; // illegal
  }
}
```

- Casting and the instanceof operator do not work properly when the object is an array whose element type is an instantiation. The reason for the problem is that Java doesn't store enough information about the element type of an array. For example consider:

```
Object aset = new BoundedSet[int][10]( );
aset[0] = new BoundedSet[int]( );
...
if (aset instanceof Collection[int]) ...; // false
BoundedSet[String][ ] s = (BoundedSet[String][ ]) aset; // works!
```

  All that is known at runtime is that the array contains some kind of BoundedSet. This information is sufficient to make the instanceof operator return false (since BoundedSet isn't declared to be a subtype of Collection). However, the cast will succeed.

  To avoid erroneous casts, a good rule is to avoid casting to array types when the element type is an instantiation unless you can prove that if the cast succeeds it is correct.

A Methods of Primitive Types

.

A.1 Methods for All Types

Primitive and array types have all the public Object methods, e.g., toString, hashCode. They also have equals and clone methods with the following signatures (in these definitions, T stands for the primitive type or array type):

```
Object clone( )
  // effects: Returns a copy of this. Since primitive types are immutable,
  //   clone is an identity function for them. For arrays, it performs
  //    a shallow copy of the elements.
```

```
boolean equals(Object obj)
  // effects: Returns true iff this is indistinguishable from obj
  //    now and in the future.

boolean equals(T obj)
  // effects: Returns true iff this is indistinguishable from obj
  //    now and in the future.
```

A.2 Methods for Arrays

All Java array types Element[ ] have the following methods:

```
Array(Array a)
  // effects: Makes this have the same elements as a.

Array(int n)
  // effects: Makes this contain n elements, with the
  //   appropriate default value for Element.

int size( )
  // effects: Returns the size of this.

Element get(int i) throws IndexOutOfBoundsException
Element elementAt(int i) throws IndexOutOfBoundsException
  // effects: If i is in bounds, returns the ith element of this
  //   else throws IndexOutOfBoundsException.

void set(int i, Element e) throws IndexOutOfBoundsException
void setElementAt(Element e, int i) throws IndexOutOfBoundsException
  // effects: If i is in bounds, stores e at this[i]; else,
  //   throws IndexOutOfBoundsException.
```

A.3 Numeric Types

The numeric types (byte, short, int, long, float, double, and char) have the following methods. In the method signatures below, "Numeric" should be replaced by the appropriate numeric type.

```
boolean equals(long l)
  // effects: Returns true iff this, when cast to a long, is equal to l.

boolean equals(double d)
  // effects: Returns true iff this, when cast to a double, is equal to d.

byte byteValue( )
short shortValue( )
int intValue( )
long longValue( )
```

```
float floatValue( )
double doubleValue( )
  // effects: Returns the value of this, cast to the appropriate type.

char charValue( )
  // effects: Returns the character equivalent to this. This method is
  //   only provided for the types byte, short, int, and char.

Numeric max(Numeric n)
  // effects: Returns the larger of this and n.

Numeric min(Numeric n)
  // effects: Returns the smaller of this and n.

int signum( )
  // effects: Return −1 if this is less than 0, 1 if this
  //   is greater than 0, and 0 if this is equal to 0.

Numeric add(Numeric n)
Numeric subtract(Numeric n)
Numeric multiply(Numeric n)
Numeric divide(Numeric n)
  // effects: Returns the sum (difference, product, quotient) of this and n.

Numeric abs( )
  // effects: Return the absolute value of this.

boolean less (Numeric n)
  // effects: Returns true if this < n else returns false.

boolean greater (Numeric n)
  // effects: Returns true if this > n else returns false.

int compareTo(Numeric n)
  // effects: Returns −1 if this < n, 1 if this > n, and 0 if this = n.
```

Numeric types also have the following constructors:

```
Numeric( )
  // effects: Initializes this to that type's default value.

Numeric(Numeric n)
  // effects: Initializes this to n.
```

In addition, all the numeric types except char have:

```
Numeric negate( )
```

// **effects**: *Returns the inverse of this.*

A.4 Logical Types

All types to which the logical operators apply (byte, short, int, long, char and boolean) have the following methods. (These methods are bitwise operations for all types but boolean, for which they are logical operations.)

Logical and(Logical b)
  // **effects**: *Returns the* and *of this and* b.

Logical or(Logical b)
  // **effects**: *Returns the* or *of this and* b.

Logical xor(Logical b)
  // **effects**: *Returns the* xor *of this and* b.

Logical not( )
  // **effects**: *Returns the* not *of this.*

A.5 Integer

All types whose values can be interpreted as integers (byte, short, int, long, and char) have the following method:

Integer mod(Integer i)
  // **effects**: *Returns this modulo* i.

A.6 Boolean

The type boolean has the following constructors and methods:

boolean( )
  // **effects**: *Makes this have the value* false.

boolean(boolean b)
  // **effects**: *Makes this have value* b.

boolean booleanValue( )
  // **effects**: *Returns a boolean value equal to this.*

References

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, October 1995. ACM SIGPLAN Notices 30(10).

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.