Increasing Java's expressiveness with `ThisType` and match-bounded polymorphism *
DRAFT

Kim B. Bruce
Williams College
Williamstown, MA 01267
kim@cs.williams.edu

October 12, 1997

In this document we would like to propose some relatively minor additions to Java's syntax and semantics [GJS96] in order to increase the expressiveness of the language with little cost in semantic complexity. We avoided making suggestions that would invalidate or change the semantics of programs written in the current version of Java.

While the main point of the language extensions are to support parametric polymorphism, we also added features which provide better support for binary methods, a kind of method that is often difficult to support in a statically typed language. (See [BCC$^+$95] for an extensive discussion of the difficulties of statically type-checking binary methods.)

Briefly, the changes proposed are to add a "`ThisType`"[1] construct to represent the type of "`this`" (the name of the object executing a method), to provide a mechanism for the programmer to specify that a variable hold values of exactly the type given (and not a type that extends it), to reinterpret "extends" as "matching" rather than "subtyping", and to add a form of constrained polymorphism which depends on matching. A more detailed discussion of these constructs and the motivations behind introducing them in typed object-oriented languages can be found in the paper [Bru97]. A complete language design including these features can be found in [BFP97].

1 Introducing constrained parametric polymorphism

There are many circumstances in which type[2] parameters are useful. For example, when one defines an array, stack, queue, or tree, the definitions are written uniformly in the type of value actually stored in the data structure. We propose supporting this by adding an optional type parameter (surrounded by "<>") after interface or class definitions.

We illustrate with a linked list example[3].

*This research was partially supported by NSF grant CCR-9424123.

[1]In my previous work, `ThisType` was given the name `MyType`. It was changed here to better correspond to the name `this` used in Java. In Eiffel, the construct `like Current` plays a similar role.

[2]We will use the word type in this proposal to include class interfaces.

[3]We are not completely wedded to the syntax used here. The fact that it is close to what is used for templates in

```
interface PolyListIfc<T> {
   public void addHead(T newElt);
   public T removeHead();
}

interface NodeIfc<T> {
   T getValue();
   NodeIfc<T> getNext();
   void setVal(T newVal);
   void setNext(NodeIfc<T> newNode);
}

public class Node<T> implements NodeIfc<T>{
   protected T value;
   protected NodeIfc<T> next;
   public Node<T>(T value, NodeIfc newNext){...}
   public void setNext(NodeIfc<T> newNode){...}
   public NodeIfc<T> getNext(){return next;}
   public T getValue(){return value;}
}

public class PolyList<T> implements PolyListIfc<T>{
   protected NodeIfc<T> head;
   public PolyList<T>(){head = null;}
   public void addHead(T newElt){...}
   public T removeHead(){...}
}
```

One would declare and create a variable `aList` of type `PolyListIfc<Integer>` with the statement:

```
PolyListIfc<Integer> aList = new PolyList<Integer>();
```

While this goes a long way toward supporting the use of parameterized types, there are important and interesting examples where we desire parameterized classes and interfaces, but need restrictions on the types to which they may be applied.

A simple variant of the previous example would involve ordered lists. Let us suppose that the ordering of the elements is determined by an integer key which is obtained via a getKey() method. Then an element of type `T` may be used as a value as long as its type, `T`, supports a `getKey` method returning an integer.[4] We specify this as follows:

```
interface Keyed{
   public int getKey();
}
```

C++ might be an advantage. On the other hand we would also be comfortable using the keyword "generic", which is already reserved in Java.

[4]We present a more general variant of ordered lists later in the paper.

```
interface PolyOrdListIfc<T extends Keyed> {
   public void insert(T newElt);
   public T removeHead();
}

public class PolyOrdList<T extends Keyed> {
   protected NodeIfc<T> head;
   public PolyOrdList<T>(){head = null;}
   public void insert(T newElt){...
            if (current.getValue().getKey() < newElt.getKey())
               ... newNode = new Node<T>(newElt,current.getNext()) ...}
   public T remove(int key){...}
}
```

Notice that we use the same `Node` class and `NodeIfc` interface as above. The fact that `T` extends `Keyed` guarantees that elements of type `T` understand the `getKey()` method.

The support for constrained genericity works fine for many circumstances, but we will see below that there are many important examples in which this system is not as expressive as we would like. We begin this exploration by examining the type of the keyword `this` in Java.

2 Exact types

Normally in an object-oriented language, if an expression has type `T`, at run time its value may be of any type which extends `T`. Sometimes, however, we want to have homogeneous data structures (i.e., ones where all values included have exactly the same type). In order to allow the programmer to specify this we introduce "exact" types. If `T` is an interface, then declaring a variable to have type `@T` indicates that at run time, the public interface of the variable will be exactly the features in `T`.

For example we could rewrite the interface and class declarations for nodes in the previous section as:

```
interface NodeIfc'<T> {
   T getValue();
   @NodeIfc'<T> getNext();
   void setVal(T newVal);
   void setNext(@NodeIfc'<T> newNode);
}

public class Node'<T> implements @NodeIfc'<T>{
   protected T value;
   protected @NodeIfc'<T> next;
   public Node'<T>(T value, NodeIfc'<T> newNext){...}
   public void setNext(@NodeIfc'<T> newNode){...}
   public @NodeIfc'<T> getNext(){return next;}
```

```
   public T getValue(){return value;}
}
```

For example the definitions of the class `Node'<T>` promises that its public interface is exactly `NodeIfc'<T>`, that the `next` field has type exactly `NodeIfc'<T>`, and that the parameter of `setNext` and return value of `getNext` are both of type exactly `NodeIfc'<T>`. This would be useful if we wanted to guarantee that all nodes of a linked list are all of exactly the same type. (This will be relevant when we introduce features which will allow us to define `DbleNode` as a subclass of `Node` in later sections. We definitely do not want to mix singly and doubly-linked nodes in the same list.) Note that the values stored in the nodes are of any type which extends `T`. Thus this example provides a good example of where one might want some items with exact types and others with more flexible types.

Programmers used to traditional imperative languages like Pascal or C are likely more comfortable and familiar with exact types. Object-oriented languages have typically only offered the more flexible types. It is worth noting that Ada 95[Int95] also allows the programmer to specify where exact or flexible types are desired. Ada 95's conventions are the `T` stands for an exact type while `T'class` represents a type including values whose types extend `T`. Our earlier language, LOOM[BFP97], uses a convention similar to that of Ada 95 in which we mark the flexible types.

Aside from the intrinsic value of allowing the programmer to specify more completely the values in the program, we will see that the use of "exact" types helps deal with problems with typing so-called binary methods after we introduce the new `ThisType` type expression.

3 Introducing `ThisType`

The keyword `this` is used in Java in method bodies of classes to represent the object executing the method. Suppose `this` occurs in the body of method `m` of class `C` and that `D` is a derived class of `C` which does not override `m`. Notice that when `m` is defined as part of `C`, `this` implicitly has type (class) `C`, whereas in the same (inherited) method of `D`, `this` has type `D`.

Because subclasses in Java also generate subtypes, this implicit change of type does not cause type errors. However it does cause the loss of information. In particular if a method returns `this` or a clone of `this`, the type system will treat the value returned as being of type `C`, even though, when executed by an object of type `D`, it will return an object of type `D`. Thus, while the type system is safe, it does lose information.

Trying to define doubly-linked nodes by extending nodes illustrates some of the problems that may arise:

```
interface DbleNodeIfc<T> extends NodeIfc<T>{
   DbleNodeIfc<T> getPrevious();
   void setPrevious(DbleNodeIfc<T> newNode);
}

public class DbleNode<T> extends Node<T> implements DbleNodeIfc<T>{
   protected DbleNodeIfc<T> previous;
   public DbleNode<T>(T value, NodeIfc<T> newNext, DbleNodeIfc<T> newPrev)
         {...}
```

```
    public void setPrevious(DbleNodeIfc<T> newNode){previous = newNode;}
    public DbleNodeIfc<T> getPrevious(){return previous;}
}
```

While the inherited code works fine, there are problems with the typing. For instance, `getNext()` returns a value of type `NodeIfc<T>` rather than `DbleNodeIfc<T>`, as desired. While this can be fixed with a type cast after each message send, the case of the method `setNext` is more problematic. When `setNext` is sent to a `DbleNode` object, it must take a parameter of type `NodeIfc<T>`, rather than `DbleNodeIfc<T>`. However attaching a singly-linked node to a doubly-linked node is almost certainly a mistake. Thus the body of the `setNext` method would have to be overridden to include a test to see if the parameter really is a doubly-linked node, with a type cast applied if it is one, and an exception raised otherwise. Interestingly, aside from the type problems, the inherited code itself performs exactly the operations desired. In particular, if one rewrote the code for these methods, it would look exactly the same aside from the type information.

We can give a more precise typing to this extended interface and subclass if we introduce the keyword `ThisType` to represent the type or interface of `this`.[5] Now we can write more accurate types for fields (both instance variables and methods) of classes.

For a first simple example, suppose we are given:

```
interface Clonable{
    @ThisType clone();
}
```

Thus any class implementing Clonable will have a method `clone` which will return an object of exactly the same type as the receiver. Thus if `x` is declared to have type `@T`, where `T` extends `Clonable`, then `x.clone()` is an expression of type `@T`. This follows from our description of the meaning of `ThisType`. Since the method *clone* returns a value of exactly the same type as the receiver, if the receiver has type `@T`, then the result has type `@T`.

If all we knew was that `x` had type `T` (not an exact type), then all we know about the value of `x` is that it has an interface which extends that of `T`. Since `clone` promises to return something with exactly the same interface as the receiver, we know that `x.clone()` will return a value whose interface extends `T`. Thus `x.clone()` is an expression with type `T`, not `@T`.[6]

For our next example we introduce a revised interface and class definition for the singly-linked node example discussed earlier.

```
interface ThisNodeIfc<T> {
    T getVal();
    @ThisType getNext();
    void setVal(T newVal);
    void setNext(@ThisType newNext);
}

class ThisNode<T> implements @ThisNodeIfc<T>{
```

[5] We emphasize that `ThisType` represents the interface of `this`, not its class!

[6] We will see below that the typing is not as simple if `ThisType` occurs as the type of an argument rather than a return type.

```
   protected T val;
   protected @ThisType next;
   public ThisNode<T>(T value, ThisNodeIfc<T> newNext){...}
   public T getVal() {return val;}
   public @ThisType getNext() {return next;}
   public void setVal(T newVal) {val = newVal;}
   public void setNext(@ThisType newNext) {next = newNext;}
}
```

The interface and class defined above are essentially equivalent to those defined earlier. Because `ThisType` stands for the type of `this`, when we send a message to an object, all occurrences of `ThisType` are replaced by the type of the receiver. In particular, if `nd` has type `@ThisNodeIfc<Integer>`, then `nd.getNext()` will return a value of type `@ThisNodeIfc<Integer>`. Similarly, the statement `nd.setnext(aNewNext)` will be well-typed only if the actual parameter has type `@ThisNodeIfc<Integer>`.

Binary methods are methods which have a parameter whose type involves `ThisType`. In the above example, `setNext` is a binary method. As will be discussed in the next section, typing rules are a bit more complicated with binary methods if the receiver has a type which is *not* exact. On the other hand, the typing of a message send of `getNext` to an expression whose argument is a non-exact type is just like our earlier example. Thus if `nd` is declared to have type `N`, where `N` extends `ThisNodeIfc<Integer>`, then `nd.getNext()` has type `N`.

The advantage of using of `ThisType` in place of `NodeIfc` is made clear when we define a derived class to represent doubly-linked nodes:

```
interface ThisDbleNodeIfc<T> extends ThisNodeIfc<T>{
   @ThisType getPrev();
   void setPrev(@ThisType newPrevious);
}

class ThisDbleNode<T> extends ThisNode<T> implements
                                          @ThisDbleNodeIfc<T>{
   protected @ThisType previous;
   public @ThisType getPrev() {return previous;}
   public void setPrev(@ThisType newPrevious) {previous = newPrevious;}
   public void setNext(@ThisType newNext){
      super.setNext(newNext);
      if (newNext != null)
         newNext.setPrevious(self);
   }
}
```

Notice that the flexible meaning of `ThisType` ensures that the type of `next` as well as the types of the result of `getNext` and the parameter of `setNext` *change* to reflect the different meaning of `this` when we move from the superclass to the subclass. For example, if `dnd` has type `DbleNode`, then `dnd.getNext()` returns a value of type `DbleNode`. Compare this with the discussion above where the same message is sent to a node.

The use of `@ThisType` rather than `@NodeIfc` as the type of `next` and in the signature of `getNext` and `setNext` methods ensures that the `ThisDbleNode` definition would reflect our design goals, and detects errors such as setting the `next` field of a doubly-linked node to a singly-linked node. It also relieves the programmer of the burden of using type-casts in order to use the value as a doubly-linked node.

Of course, in our proposed extension it will still be possible to replace all occurrences of `ThisType` above by an explicit node type like `NodeIfc` if that is what is desired, however with `ThisType` the user has the option to decide which is most appropriate. In most cases we expect that it will be `ThisType`.

It is worth noting that some other languages also support constructs like `ThisType`. The language Trellis/Owl includes a `MyType` construct to stand for the type of `me` (it's equivalent of `this`), while Eiffel uses `like Current` for the type of `Current`. In Trellis/Owl, however, subclasses must always generate subtypes, so binary methods may not appear in superclasses, limiting the expressiveness relative to our proposal. Eiffel, on the other hand, does not type-check this construct properly, allowing the user to write, compile, and run code like that given in our `breakit` example in the next section.

Ada 95 also supports the equivalent of `ThisType`, though without adding a new keyword. In Ada, if one defines a `Node` class in which the methods or fields mention `Node`, then in subclasses, all occurrences of `Node` in the methods or fields are implicitly replaced by the subclass name. Thus in subclass `DbleNode`, the type of the `next` field would automatically be changed to `DbleNode`. There are two drawbacks to this construct in Ada 95. One is that the programmer will likely be surprised by this change, and in some cases might not want it to occur. The other is that Ada does not type check this construct so that it is guaranteed to work properly in subclasses. As a result the system must insert dynamic checks.

Our earlier languages, TOOPLE, PolyTOIL, and LOOM all included a feature essentially the same as that proposed here.

4 Replacing Subtyping by Matching

One drawback to the introduction of `ThisType` is that subclasses (derived classes) need no longer generate subtypes. For example, the interface `ThisDbleNodeIfc` is not a subtype of `ThisNodeIfc`. Thus, one may not use a value of type `ThisDbleNodeIfc` in some contexts which expect a value of type `ThisNodeIfc`. The program `breakit` illustrates the problem.

```
public void breakit(ThisNodeIfc nd1, ThisNodeIfc nd2)
{
   nd1.setNext(nd2);
}
```

If `aNd1` and `aNd2` are both nodes, then the call of `breakit(aNd1,aNd2)` would execute without problem. Suppose however that `aDnd1` is a doubly-linked node, while `aNd2` is a singly-linked node. Then the call `breakit(aDnd1,aNd2)` will not be safe because the message `setNext` will be sent to a doubly-linked node with a parameter which is a singly-linked node, while the typing rules require that the parameter type be the same as the receiver. In particular, if this call were executed, a crash would result when `aNd2` is sent the message `setPrevious`. Thus `DbleNodeIfc` cannot be a subtype of `NodeIfc`.

This failure of subtyping arises whenever the original type (interface) has a binary method. In those cases the subclass does not generate a subtype. However, there is another relationship, called "matching", between interfaces that is perhaps even more useful than subtyping.

Matching, as defined in the language LOOM [BFP97] corresponds exactly to extension of interfaces when we have type `ThisType`. As a result all uses of `extends` between interfaces will now correspond to matching. (Recall, however, that in the special case that the type being extended has no binary methods, then *extends* also corresponds to subtyping.)

Subtyping was useful because if `S` is a subtype of `T`, then a value of type `S` could be used in any context that expected a value of type `T`. As we have seen, `extends` no longer corresponds to subtyping. Of what use is it to know that one type extends another? The key is that if `S` extends `T`, for `T` a known type, then this provides us with information on what messages can be sent to an object of type `S`.

For example suppose that interface `FNodeIfc` extends `ThisNodeIfc`, where `ThisNodeIfc` is as defined in the previous section. If `fnd` is a value of type `@FNodeIfc` then, because `FNodeIfc` extends `ThisNodeIfc`, `fnd.getNext()` is defined and will return a value of type `@FNodeIfc`. This follows because the `extends` relation between interfaces implies any value of type `FNodeIfc` must have a parameterless method `getNext` that returns a value of type `ThisType`. Similarly we can determine that `fnd.setNext(newFnd)` is defined and is well-typed as long as the value of `newFnd` has type `@FNodeIfc`.

What happens to a message send involving `ThisType` if we don't know the exact type of the receiver? That is, if `gnd` has type `ThisNodeIfc`, then the value of `gnd` at run-time may have any type which extends Node. As we saw earlier, in the case of a method like `getNext()` which has return type `@ThisType`, there is no problem. At runtime the variable `gnd` will have a value of some type matching `ThisNodeIfc`, call it `GNodeIfc`. Because `getNext` returns a value of type `ThisType`, `nd.getNext()` will return a value of type `GNodeIfc`, which, by assumption matches `ThisNodeIfc`. Thus we can deduce that `nd.getNext()` has type `ThisNodeIfc`, indicating that the value is of some type matching `ThisNodeIfc`.

However we have more serious problems if the method has a parameter with type involving `ThisType`. Look again at `gnd.setNext(newGnd)`. Because `setNext` takes a parameter of type `@ThisType`, the type of the value of `newGnd` must be the same as the type of `gnd`. Thus if `gnd` has a value of type `@NodeIfc`, then the value of `newGnd` must be of type `@NodeIfc`. On the other hand if the value of `gnd` actually has type `@DbleNodeIfc`, then the value of `newGnd` must be of type `@DbleNodeIfc`. Since we do not know statically the exact type of `gnd`, we cannot determine statically what type the parameter must have to guarantee that the call will be type safe. Therefore we must rule out sending "binary messages" (i.e., messages corresponding to binary methods) to objects whose exact type is unknown. In other words, binary messages can only be sent to expressions whose type is of the form `@T`.

Using this information we see that the definition of the method `breakit` at the beginning of this section does not type check since we do not know the exact type of the receiver, `nd1`, of the `setNext` message, yet this is a binary method. However if we rewrite the example with exact types there is no problem:

```
public void nobreakit1(@ThisNodeIfc nd1, @ThisNodeIfc nd2)
{
   nd1.setNext(nd2);
}
```

A more general rewriting gives:

```
public void nobreakit2<N extends ThisNodeIfc>(@N nd1, @N nd2)
{
   nd1.setNext(nd2);
}
```

In either case the method type checks without difficulty since we know the exact type of the receiver. In the definition of `nobreakit1` the specification of an exact type ensures that we canonly invoke it with a parameter which is a singly-linked node. In the case of `nobreakit2`, the parameters can be of any type, `N`, extending `ThisNodeIfc`, but both parameters must be of exactly that type.[7]

Despite the restriction that the target of binary message must have an exact type, we will see that in many (perhaps even most) circumstances, matching is more useful than subtyping. We illustrate this in the next section where we revisit bounded polymorphic (or generic) classes where we take advantage of `ThisType`.

5 Constrained polymorphism with `ThisType`

In the ordered list example in section 1, we constrained the type parameter to include a method `getKey` which returned an integer. We could then keep the list sorted by comparing the keys of items. While this was effective for that example, it is more restrictive than one would like. In general one would like to instead require that the type support ordering relations. We will see that this is relatively straight-forward using `ThisType`.

Our first example uses the node interfaces and classes from section 3 to construct a class which can generate either singly or doubly-linked lists holding values of type `T`.

```
public class List<T, N extends ThisNodeIfc<T>>{
   protected @N head;
   public List<T,N>(){head = null;}
   public void addToHead(@N newNode){
      newNode.setNext(head);
      head = newNode;}
   public T deleteHead(){
      T temp = head.getVal();
      head = head.getNext();
      return temp;}
   ...
}
```

The expression `new List<Integer, ThisNodeIfc<Integer>>` creates an empty singly-linked list capable of holding `Integers`, while `new List<String, ThisDbleNodeIfc<String>>` creates an empty doubly-linked list capable of holding `Strings`.[8]

[7] While we will not mention it further, it is possible to extend the language so that both functions and classes take type parameters.

[8] We suggest adding a `typedef` construct in the programs in the appendices to allow the use of names to stand for more complex type definitions, making declarations like these easier to read.

Note that lists formed by instantiating `List` with type `T` hold elements of any type extending `T`. That is, it generates heterogeneous lists. Adding a few "`@`"s to appropriate occurrences of `T` in the node classes and `List` would result in homogeneous lists. (See Appendix B for more details.)

When we define ordered structures we need to be able to compare elements. The following interface lists the necessary operations.

```
interface OrdEltType{
   boolean gt(@ThisType other);
   boolean eq(@ThisType other);
}
```

We can now form a (homogeneous) ordered data structure using any type `T` which matches `OrdEltType`, since if two elements, `o` and `p` have type `@T`, then `o.gt(p)` will be defined and type-correct. The example in Appendix B of polymorphic homogeneous ordered lists uses a type variable which extends `OrdEltType`. Notice that this is more general than our previous example in section 1 using the class `Keyed` as a constraint.

The fact that **extends** now corresponds to matching was quite important for both of these examples, as there are no non-trivial subtypes of either `OrdEltType` or `ThisNodeIfc<T>` because of the occurrences of `ThisType` as a parameter in at least one method of each. Thus matching is essential in describing the constraint on the type parameter.

Generic subclasses are also easy to write with this setup. For example, the following defines an ordered list class which keeps track of a special "current" element.

```
class CurList <T extends NewBound,N extends ThisNodeIfc<T>> extends List<T,N> {
    @N current;...}
```

Note that in the above example we made the bound of `T` smaller (in the "extends" ordering) than the one for `List`. The only requirement is that the new bound extend the bound in the superclass.

In Appendix B we present three more examples of complete programs for variants of ordered linked lists which illustrate the flexibility of these constructs. As in the example above, these list classes can be instantiated with either singly or doubly-linked nodes, so that the same class can be used to create singly or doubly-linked lists. Both homogeneous and heterogeneous lists are defined there.

It should be possible to instantiate type parameters with classes rather than just interfaces as we have done in all the examples, though we have not yet investigated this in detail. Our own philosophical inclination is that users should define interfaces for all classes, just as they do for non-object values (though we realize that others might not agree). Thus our own practice is to only use interfaces.

It is worth noting that since objects are implemented as references, there is no difficulty in compiling generic routines and then using the same code in instantiations as long as the type parameters are guaranteed to be interfaces or classes.

6 Comparison with other proposals

In this section we provide a brief comparison of our proposal for extending Java with those given by Agesen et al. [AFM97], Odersky and Wadler [OW97], Myers et al. [MBL97], and Thorup [Tho97].

The main thing to point out here is that while the F-bounded subtyping presented in [OW97] and [AFM97], and the *where* clauses in [MBL97] provide essentially the same constraints as matching, the mismatch between subtyping and inheritance creates anomolies. We illustrate these with the following example from [OW97]:

```
Interface Ord<elem>{
   boolean less(elem o);
}

class Pair<elem implements Ord<elem>>{
   protected elem x; protected elem y;
   public Pair(elem x, elem y){this.x = x; this.y = y;}
   public elem min(){if (x.less(y)) return x; else return y;}
}

class OrdInt implements Ord<OrdInt>{
   protected int i;
   public OrdInt(int i){this.i = i;}
   public int intValue(){return i;}
   public boolean less(OrdInt o){return i < o.intValue();}
}

Pair<OrdInt> p = new Pair(new OrdInt(22),new OrdInt(64));
System.out.println(p.min().intValue());
```

So far, so good. But suppose we now define a subclass of OrdInt:

```
class ColorOrdInt extends OrdInt{
   protected Color c;
   public ColorOrdInt(int i, Color c){super(i); this.color = color;}
   public Color getColor(){return color;}
}
```

One might expect that `ColorOrdInt` would also be usable with `Pair`, but `ColorOrdInt` does *not* implement `Ord<ColorOrdInt>` because the `less` method takes a parameter of type `OrdInt` rather than `ColorOrdInt`. Therefore `Pair`<`ColorOrdInt`> would not be allowed by the type system. Note that exactly the same problem arises if we had used *where* clauses to constrain the type parameter to `Pair`.

We can rewrite this example in our proposed extension of Java as follows:

```
Interface Ord{
   boolean less(@ThisType o);
}

class Pair<elem extends Ord>{
   protected @elem x; protected @elem y;
```

```
   public Pair<elem>(@elem x, @elem y){this.x = x; this.y = y;}
   public @elem min(){if (x.less(y)) return x; else return y;}
}

interface OrdIntIfc extends Ord{
   int intValue();
}

class OrdInt implements @OrdIntIfc{
   protected int i;
   public OrdInt(int i){this.i = i;}
   public int intValue(){return i;}
   public boolean less(@ThisType o){return i < o.intValue();}
}

@Pair<OrdIntIfc> p = new Pair<OrdIntIfc>(new OrdInt(22),new OrdInt(64));
System.out.println(p.min().intValue());
```

The use of `ThisType` makes writing the type constraint in `Pair` somewhat simpler than before. Because we have chosen to only allow interfaces as parameters we defined `OrdIntIfc` as the interface for `OrdInt` and used it to specialize `Pair`.

We note that our implementation of `Pair` appears to differ from that of the Pizza code by requiring the arguments of pair to be of an exact type. However, the Pizza proposal requires type parameters to be treated as exact types. Thus the Pizza code also implements homogeneous pairs.

We define the subclass:

```
interface ColorOrdIntIfc extends OrdIntIfc{
   Color getColor();
}

class ColorOrdInt extends @ColorOrdIntIfc{
   protected Color c;
   public ColorOrdInt(int i, Color c){super(i); this.color = color;}
   public Color getColor(){return color;}
}
```

Because `ColorOrdIntIfc` extends `Ord` there is no problem in using `Pair`<`ColorOrdIntIfc`>, so the objects from the subclass can be used in pairs.

Thorup's proposal[Tho97] using "virtual types" is quite different from the others cited above. While this proposal includes a construct like our `ThisType`, we ignore it for now in order to make a point.

```
public class IntNode{
   typedef N as IntNode
   protected int value;
   protected N next;
   public Node(int value, N newNext){...}
```

```
   public void setNext(N newNode){...}
   public N getNext(){return next;}
   public int getValue(){return value;}
}
```

The `typedef` clause means that all occurrences of `N` in the class definition represent `IntNode`. The point of the `typedef` clause is to allow covariant redefinition in subclasses. Thus we can define:

```
public class IntDbleNode extends IntNode{
   typedef N as IntDbleNode
   protected N previous;
   public IntDbleNode(N newPrevious, int value, N newNext){...}
   public void setPrevious(N newNode){...}
   public N getPrevious(){return previous;}
}
```

From the point of view of static type-safety, there are two problems with such redefinition. The first is that the lack of restriction on covariant changes to types defined in `typedef`'s may lead to type errors in method bodies. That is, a method body may be fine if the type variable `N` stands for `IntNode`, but may break if `N` stands for `IntDbleNode`. A second problem is that the subclass may not represent a subtype. In particular, a method similar to our `breakit` routine will work fine for `IntNode`, but break if one of its arguments is of type `IntDbleNode`.

These problems are essentially the same as those that arise with respect to static type safety in the languages Eiffel [Mey92] and Beta [KMMPN87]. Like Beta (and unlike Eiffel), Thorup's proposal suggests that both of these kinds of errors would be checked at run-time, so the language would remain type-safe. Basically this proposal trades off static type safety for greater ease of expression. We believe that there are significant advantages to static type-safety, and hence prefer the other proposals to this one.

7 Conclusion

We have proposed extending the syntax of Java to support the definition of parameterized classes and interfaces with bounded type abstractions. We have also proposed introducing the new keyword, `ThisType`, to represent the type of `this`, and a notation, `@T`, to indicate that a variable or parameter has exactly type `T`. When `ThisType` is used in the type of methods, the keyword `extends` no longer necessarily corresponds to subtyping. Instead it represents the more general *matching* relation in [BFP97]. The use of `ThisType` allows the programmer finer control over the specification of types in classes, in particular allowing controlled changes in the types of instance variables and methods in subclasses. The introduction of constrained genericity saves the programmer from copying over and making minor (but error-prone) modifications to blocks of code when the types of components changes. Also these generic classes can be type-checked once, guaranteeing that any instantiation which satisfies the constraints cannot give rise to type errors (unlike C++'s templates which are only type checked upon instantiation).[9]

[9] A technical point is that methods of class `C` are type-checked under the assumption that `ThisType extends C`. This ensures that methods remain type-correct in subclasses. See Appendix A for the details.

The only down side to these extensions is that subclasses will no longer always result in subtypes. While this may at first appear to be a big loss in expressiveness, the correspondence of `extends` to matching results instead in gains in expressiveness, in particular in the definition of subclasses and constrained genericity. Because most uses of subtyping are simply to ensure that certain methods exist, matching provides exactly what is desired.

One possible concern about using this language design is that the programmer needs to indicate whether types are to be exact types or to encompass values from matching types. As noted earlier, all current Java programs continue to work exactly as before. In this paper we have presented more difficult programs that require `ThisType` and exact types in order to demonstrate the expressiveness of the language, but most programs will use neither of these constructs. The point is that they are available when useful and there are a number of examples where binary methods are exactly what is needed, leading to the need for `ThisType` and exact types. Finally we note again that the Ada 95 also requires the programmer to decide when exact or more flexible types are used. This decision does not appear to be a burden to Ada 95 programmers.

In summary, we believe that the proposed additions to the language will greatly increase the flexibility of the Java's type system, making it easier for programmers to write correct programs in the language.

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parameterized types to Java. In *ACM Symposium on Object-Oriented programming: System, Languages, and Applications*, page to appear, 1997.

[BCC+95] 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, 1995.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[Bru97] Kim Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Williams College, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Int95] Intermetrics. *Ada 95 Reference Manual, version 6.0.* 1995.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

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

[Mey92] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *24th ACM Symp. on the Principles of Programming Languages*, pages 146–159. ACM, 1997.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, pages 444–471. LNCS 1241, Springer-Verlag, 1997.

A Informal Type-Checking Rules

Classes are type-checked as follows: If `C` is a class definition, let the "public interface" of `C` be the interface composed of the signatures of all public features of `C`. To type-check a class `C`, first form the public interface, `IC`, of the class. Now type-check all features of `C` under the assumption that `ThisType` extends `IC` and that `this` has type `@ThisType`. Accesses to instance variables (either prefixed by *this* or not) are assigned their declared types in the class, but message sends to `this` (again written either implicitly or explicitly) use the (weak) information given above on the type of `this`. As with Java now, one must check that `IC` extends any interfaces that `C` is declared to implement.

This minor modification to type-checking rules for classes ensures that inherited methods need not be type-checked again in subclasses, because the public interface corresponding to the subclass will be an extension of the public interface of the superclass, and type-checking in the superclass was performed only on the assumption that the meaning of `ThisType` was an extension of the public interface of that superclass.

There are two type-checking rules for message passing depending on whether the receiver is an exact type or not. The simplest rule is for exact types. Suppose `o` has type `@T` where `T extends T'` and `T'` has a method `m` with signature `U m(S arg)`.[10] Then `o.m` has signature `U' o.m(S' arg)` where `U'` and `S'` are formed from `U` and `S`, respectively, by replacing all occurrences of `ThisType` by `T`, the type of the receiver. For example, suppose `elt` has type `@Ord`, where `Ord` was defined in section 6 to have method `less` with signature `boolean less(@ThisType o)`. Then `elt.less(other)` will be well-typed only if `other` has type `@Ord`.

If a message is sent to a non-exact type, then one must be slightly more careful with occurrences of `ThisType`. Suppose `o` has type `T`, where `T` is not an exact type, and `T extends T'` where `T'` has a method `m` with signature `U m(S arg)` in which `S` does not contain any free occurrences of `ThisType`. Then `o.m` has signature `U'' o.m(S arg)` where `U''` is formed from `U` by replacing all occurrences of `@ThisType` by `T` and then all remaining occurrences of `ThisType` by `T`. For example if interface `T` has a `clone` method with signature `@ThisType clone()` and `o` has type `T`, then `o.clone()` has type `T`. Note that this expression should not have type `@T` since we do not know the exact type of the receiver, `o`. Sending a binary message to a non-exact type is never type-correct.

Type-checking classes (or functions) with type parameters is straight-forward. Unbounded type parameters are considered to be constrained by `Object`. To type check an expression with type parameter `T extends U`, simply type-check the rest of the expression under the assumption that `T`

[10] Note that `T` can be the same as `T'` if `T` is an interface expression, but in general `T` might be a type variable and `T'` might be its constraint. Because types of methods are not allowed to change in extensions of interfaces, so there is no ambiguity based on the choice of the particular `T'`.

extends U. Application of expressions with type parameters to actual types simply involves making sure the actual type satisfies the constraints and then replacing all occurrences of the formal type parameter by the actual type in the remaining expression.

The other type checking rules are the same as Java or are obvious extensions.

B Extended Java Examples

In this appendix we present three examples of extended Java programs. In section 5 we presented an example of a class which generates either singly or doubly-linked lists. Ordered lists are more complex because of the need for a compatible ordering relation. The first example is a relatively standard example of a polymorphic homogeneous ordered linked list class (which can be instantiated to both singly and doubly-linked lists). It uses an interface in which the ordering relations are binary methods. The next two are heterogeneous ordered linked lists. The first one has a fixed upper bound on the values stored in the list while the second has a polymorphic upper bound. Neither requires binary methods, but the second requires more sophisticated bounds on parameters than the other examples.

B.1 Polymorphic Homogeneous Ordered Linked List

This program defines a class which supports homogeous ordered lists, that is, lists in which all elements have exactly the same type. One of the advantages of the use of `ThisType` and matching is that we can write a generic ordered list class which can be instantiated as being either singly or doubly-linked.

```
interface ExactNodeIfc <T>{
   @T getValue();
   void setValue(@T newValue);
   @ThisType getNext();
   void setNext(@ThisType newNext);
}

/****************************************************************/

public class ExactNode<T> implements @ExactNodeIfc<T>{
// fields

    @T value;            // value of node
    @ThisType next;        // node to right of current node

// constructor

    public ExactNode<T>(@T value){
        this(value,null);
    }
```

```
    public ExactNode<T>(@T value, @ThisType next){
        this.value = value;
        next = this.next;
    }

// methods

    public @T getValue(){
        return value;
    }

    public void setValue(@T newValue) {
        value = newValue
    }

    public @ThisType getNext(){
        return next;
    }

    public void setNext(@ThisType newNext) {
        next = newNext;
    }
}

/**************************************************************/

interface ExactDbleNodeIfc<T> extends ExactNodeIfc<T>{
    @ThisType getPrev();
    void setPrev(@ThisType previous);
}

/**************************************************************/

public class ExactDbleNode<T> extends ExactNode<T>
                            implements @ExactDbleNodeIfc<T>{
// fields

    @ThisType previous;

// constructor

    public ExactDbleNode<T>(@ThisType previous, @T value, @ThisType next){
        super(value,next);
        this.previous = previous;
    }
```

```
    public ExactDbleNode<T>(@T value){
        this(null,value,null);
    }

// methods

    public @ThisType getPrev(){
        return previous;
    }

    public void setPrev(@ThisType newPrev){
        previous = newPrev;
    }

    // hooks newNext to right of current node
    public void setNext(@ThisType newNext) {
        super.setNext(newNext);
        if (newNext != null)
            newNext.setPrev(this);
    }
}

/*****************************************************************/

interface OrdEltIfc{
    boolean gt(@ThisType other);
    boolean eq(@ThisType other);
}

/*****************************************************************/

// A homogeneous ordered list of items of fixed type T which matches
// OrdEltIfc.  The list could be singly or doubly linked depending on
// the value of N.
interface OrdListIfc <T extends OrdEltIfc,
                       N extends ExactNodeIfc<T>>{
   boolean find(@T match);
   void addNode(@N newNode);
}

/*****************************************************************/

public class OrdList <T extends OrdEltIfc, N extends ExactNodeIfc<T>>
                                            implements OrdListIfc<T,N>{
```

```
    protected @N head = null;        // head of list

    // Determine if match is contained in list
    public boolean find(@T match){
        @N current = head;
        while (current != nil) && match.gt(current.getValue())
            current = current.getNext();
        return (current != nil) && (current.getValue()).eq(match));
    }

    // Add a node to the ordered list
    public void addNode(@N newNode){
        if (head == null)            // list is empty
            head = newNode;
        else{                        // list is non-empty
            @N current = head;
            @N prevNode = null;
            while ((current != null) &&
                    newNode.getValue().gt(current.getValue())){
                prevNode = current;
                current = current.getNext();
            }
            if (prevNode == null){    // new element should be new head
                newNode.setNext(head);
                head = newNode;
            } else {                 // insert after head
                newNode.setNext(current);
                prevNode.setNext(newNode);
            }
        }
     }

}

/****************************************************************/

interface SomeOrdIfc extends OrdListIfc {
    ...
}

/****************************************************************/

public class SomeOrdClass implements SomeOrdIfc {
    ...
}
```

```
/**************************************************************/

public class Test{
    public static main(String[] args){
        OrdListIfc<SomeOrdIfc,ExactNodeIfc<SomeOrdIfc>> slist
                = new OrdList<SomeOrdIfc,ExactNodeIfc<SomeOrdIfc>>();
                // singly-linked homogeneous list

        OrdListIfc<SomeOrdIfc,ExactDbleNodeIfc<SomeOrdIfc>> dlist
                = new OrdList<SomeOrdIfc,ExactDbleNodeIfc<SomeOrdIfc>>();
                // doubly-linked homogeneous list
        SomeOrdIfc elt = new SomeOrdClass(val);
        slist.addNode(new ExactNode<SomeOrdIfc>(elt));
        if (slist.find(elt))
            ...
        dlist.addNode(new ExactDbleNode<SomeOrdIfc>(elt))
        if (dlist.find(elt))
            ...

    }
}
```

B.2 Heterogeneous Ordered Linked List

In this program we define heterogeneous linked lists in which all elements of the list are of a type matching some fixed type `HetOrdEltIfc`. As in the previous example, `HetOrdList` can be instantiated to be either a singly- or doubly-linked list. We introduce the new keyword `typedef` simply to make type abbreviations. We use the definitions of `ThisNodeIfc<T>`, `ThisDbleNodeIfc<T>`, `ThisNode<T>`, and `ThisDbleNode<T>` from section 3. Recall that these nodes hold values with type `T` rather than `@T`, so will support heterogeneous lists. In order to support a heterogeneous list, the methods of `HetOrdEltIfc` take arguments of type `HetOrdEltIfc` rather than `ThisType`.

```
interface HetOrdEltIfc{
    boolean gt(HetOrdEltIfc other);
    boolean eq(HetOrdEltIfc other);
    SomeOrderable getKey();        // something to be used in sorting
}

/**************************************************************/

// typedefs are not really part of Java but form useful abbreviations

typedef OrdNodeIfc = ThisNodeIfc<HetOrdEltIfc>;
```

```
typedef OrdDbleNodeIfc = ThisDbleNodeIfc<HetOrdEltIfc>;

/************************************************************/

// A heterogeneous ordered list of items all matching OrdEltIfc
// It could be singly or doubly linked depending on the value of N
interface OrdListIfc<N extends OrdNodeIfc>{
   boolean find(OrdEltIfc match);
   void addNode(@N newNode);
}


/************************************************************/

public class OrdList<N extends OrdNodeIfc> implements OrdListIfc<N>{
    protected @N head = null;        // head of list

    // Determine if match is contained in list
    public boolean find(OrdEltIfc match){
        @N current = head;
        while (current != nil) && match.gt(current.getValue())
            current = current.getNext();
        return (current != nil) && (current.getValue()).eq(match));
    }

    // Add a node to the ordered list
    public void addNode(@N newNode){
        if (head == null)            // list is empty
            head = newNode;
        else{                        // list is non-empty
            @N current = head;
            @N prevNode = null;
            while ((current != null) &&
                    newNode.getValue().gt(current.getValue())){
                prevNode = current;
                current = current.getNext();
            }
            if (prevNode == null){    // new element should be new head
                newNode.setNext(head);
                head = newNode;
            } else {                 // insert after head
                newNode.setNext(current);
                prevNode.setNext(newNode);
            }
        }
```

```
    }

}


/**************************************************************/
// Test program to try doubly and singly-linked heterogeneous lists.
public class Test{
    public static main(String[] args){
        OrdListIfc<OrdNodeIfc> slist = new OrdList<OrdNodeIfc>();
                // singly-linked heterogeneous list

        OrdListIfc<OrdDbleNodeIfc> dlist= new OrdList<OrdDbleNodeIfc>();
                // doubly-linked heterogeneous list
        OrdEltIfc elt = new SomeOrderedClass(val);  // some appropriate class
        slist.addNode(new ThisNode<HetOrdEltIfc>(elt));
        if (slist.find(elt))
            ...
        dlist.addNode(new ThisDbleNode<HetOrdEltIfc>(elt))
        if (dlist.find(elt))
            ...
    }
}
```

B.3 Polymorphic Heterogeneous Linked List

This program is a bit more complex than the previous one since we allow the user to choose the upper bound of the elements in the heterogeneous linked list. We only require that it match the interface `HetOrdEltIfc`. Note that this example requires the use of F-bounded matching in order to capture the polymorphic constraints on the heterogeneous lists.

```
interface HetOrdEltIfc<T>{
    boolean gt(T other);
    boolean eq(T other);
}

// typedefs are not really part of Java but form useful abbreviations

typedef OrdNodeIfc<T> = ThisNodeIfc<HetOrdEltIfc<T>>;
      // Node holding elements of type T, where T has gt and eq methods

typedef OrdDbleNodeIfc<T> = ThisDbleNodeIfc<HetOrdEltIfc<T>>;
      // DbleNode holding elements of type T, where T has gt and eq methods

/**************************************************************/
```

```
// A heterogeneous ordered list of items all matching type T,
// where T extends HetOrdEltIfc<T>
// It could be singly or doubly linked depending on the value of N
interface OrdListIfc <T extends HetOrdEltIfc<T>> <N extends OrdNodeIfc<T>>{
   boolean find(T match);
   void addNode(@N newNode);
}

/*************************************************************/

public class OrdList <T extends OrdEltIfc<T>,N extends OrdNodeIfc<T>>
                                            implements OrdListIfc<T,N>{
    protected @N head = null;       // head of list

    // Determine if match is contained in list
    public boolean find(T match){
        @N current = head;
        while (current != nil) && match.gt(current.getValue())
            current = current.getNext();
        return (current != nil) && (current.getValue()).eq(match));
    }

    // Add a node to the ordered list
    public void addNode(@N newNode){
        if (head == null)           // list is empty
            head = newNode;
        else{                       // list is non-empty
            @N current = head;
            @N prevNode = null;
            while ((current != null) &&
                    newNode.getValue().gt(current.getValue())){
                prevNode = current;
                current = current.getNext();
            }
            if (prevNode == null){    // new element should be new head
                newNode.setNext(head);
                head = newNode;
            } else {                // insert after head
                newNode.setNext(current);
                prevNode.setNext(newNode);
            }
        }
     }
```

```
}

/***********************************************************/

interface SomeOrdIntface extends OrdListIfc<SomeOrdIntface> {
    ...
}

/***********************************************************/

public class SomeOrdClass implements SomeOrdIntface {
    ...
}

/***********************************************************/

typedef SomeOrdNodeIfc = OrdNodeIfc<SomeOrdIntface>;
typedef SomeOrdDbleNodeIfc = OrdDbleNodeIfc<SomeOrdIntface>;

public class Test{
    public static main(String[] args){
        OrdListIfc<SomeOrdNodeIfc> slist
                            = new OrdList<SomeOrdIntface,SomeOrdNodeIfc>();
                // singly-linked heterogeneous list

        OrdListIfc<SomeOrdDbleNodeIfc> dlist
                            = new OrdList<SomeOrdIntface,SomeOrdDbleNodeIfc>();
                // doubly-linked heterogeneous list
        SomeOrdIntface elt = new SomeOrdClass(val);
        slist.addNode(new ThisNode<SomeOrdIntface>(elt));
        if (slist.find(elt))
            ...
        dlist.addNode(new ThisDbleNode<SomeOrdIntface>(elt))
        if (dlist.find(elt))
            ...

    }
}
```