A statically safe alternative to virtual types[*]

Kim B. Bruce[**1], Martin Odersky[2], and Philip Wadler[3]

[1] Williams College, Williamstown, MA, USA,
kim@cs.williams.edu, http://www.cs.williams.edu/~kim/

[2] University of South Australia,
odersky@cis.unisa.edu.au, http://www.cis.unisa.edu.au/~cismxo/

[3] Bell Labs, Lucent Technologies,
wadler@research.bell-labs.com, http://www.cs.bell-labs.com/~wadler/

Abstract. Parametric types and virtual types have recently been proposed as extensions to Java to support genericity. In this paper we investigate the strengths and weaknesses of each. We suggest a variant of virtual types which has similar expressiveness, but supports safe static type checking. This results in a language in which both parametric types and virtual types are well-integrated, and which is statically type-safe.

Keywords: Language design, virtual types, parametric polymorphism, static type checking

1 Introduction

The first step to a good answer is a good question. This note raises (and suggests an answer to) the question: Can the best features of parametric types and virtual types be integrated?

Parametric types and virtual types have both been proposed as extensions to Java, and address roughly similar issues. Parametric types date back to arrays in Fortran, with key contributions to their theory coming from Strachey [Str67] and Reynolds [Rey74,Rey83], and examples of practice appearing in languages as diverse as Ada (generics), C++ (templates), and Standard ML (parametric polymorphism). Parametric types for Java have been proposed by Myers, Bank, and Liskov [MBL97], by Agesen, Freund, and Mitchell [AFM97], by Bruce [Bru97], and by Odersky and Wadler as part of Pizza [OW97,OW98]. Virtual types first appeared in Beta [KMMN83,MM89,MMN93] and have been proposed for Java by Thorup [Tho97].

Parametric types and virtual types have complementary strengths. Parametric types are especially useful for collection types, such as lists or sets. (Users of C++ templates and collection classes will know many examples [KM96].) Virtual types are especially useful for families of types, such as the Subject/Observer family. (Users of design patterns will know many examples [GRJV94].)

The first step to a good question is a good example. This note presents two paradigmatic examples. The first, zip, demonstrates the strengths of parametric

[*] To appear, ECOOP 98

[**] Kim Bruce's research was partially supported by NSF grant CCR-9424123.

types. The second, lists with lengths, demonstrates the strengths of virtual types; we illustrate the important case of mutual recursion by extending this example to alternating lists. In each case, the program structure made easy by one approach can be mimicked by the other, but with difficulty.

In an attempt to bring together the strengths of each of these techniques, we present a variant of the virtual types notation that is statically type-safe. The new notation generalizes the `MyType` notation of Bruce and others to apply to mutually recursive types. This notation can be considered as a variant of the proposal for handling inheritance in the presence of mutual recursion put forward by Palsberg and Schwartzbach [PS94].

The ideas reported here gestated in the deliberations of a mailing list organized by Ole Agesen to discuss extensions of Java that support generic types. Gilad Bracha, Corky Cartwright, Guy Steele, Kresten Thorup, and Mads Torgersen made particularly significant contributions.

The rest of the paper is organized as follows. Section 2 reviews parametric types and virtual types. Section 3 presents examples which are easy to express with parametric types, but are hard to express with virtual types. Section 4 presents an example that is easy to express with virtual types, but hard to express with parametric types. To highlight comparisons we write each example using both notations.

Section 5 proposes a notation that is similar to virtual types, but which can be statically type checked. Section 6 shows how this notation combines with parametric types in a smooth way. Section 7 concludes.

Appendix A reworks an example of Thorup's based on the Observer pattern.

2 Parametric and virtual types

We briefly review the extension of Java to include parametric types as in Odersky and Wadler's Pizza [OW97,OW98], and virtual types as in Thorup's proposal [Tho97]. Both extensions are defined by their translation into ordinary Java.

2.1 Parametric types

Here is a simple program using parametric types.

```
public class Cell<A> {
  protected A value;
  public Cell (A v) { value = v; }
  public void set (A v) { value = v; }
  public A get () { return value; }
}
public class Test {
  public static void main (String[] args) {
    Cell<String> c = new Cell("even");
    c.set("s"+c.get());
    Cell<Object> d = c;  // compile-time error
```

```
    d.set(new Integer(7));
  }
}
```

An object of class `Cell<A>` has a protected `value` field containing an object of type `A`, which is updated by the `set` method and retrieved by the `get` method. The test code shows a trivial use of this class: creating a cell with a string value, then updating the cell to append a string to the front of its contents.

The test code also shows a trivially wrong use: trying to put an integer in a cell that should contain a string. Since type parameters are not maintained at run-time, this error must be detected at compile time. To allow this, the type system must be quite strict, prohibiting a `Cell<String>` from being assigned to a `Cell<Object>`. (Compare this will Java arrays, where it is possible to assign a `String[]` to an `Object[]`, but this requires that each array carries its type with it at run-time, and that a run-time type check is performed each time an array element is updated.)

If we omit the last two statements of the `main` method, then the code above is type-safe. It is equivalent to the following code in Java without parametric types. (Odersky and Wadler [OW97] describe two different translations of parametric types into Java, homogeneous and heterogeneous. We use the former here.)

```
public class Cell {
  protected Object value;
  public Cell (Object v) { value = v; }
  public void set (Object v) { value = v; }
  public Object get () { return value; }
}
public class Test {
  public static void main (String[] args) {
    Cell c = new Cell("even");
    c.set("s"+(String)c.get());
  }
}
```

All type parameters have been erased, and all occurrences of the parameter `A` within `Cell` have been replaced by `Object`. An appropriate cast has been added to the call to `get`, and one can guarantee at compile-time the added cast will never fail at run-time.

The final code closely resembles an idiom often used by Java programmers to mimic the effect of parametric types. For instance, this idiom is used extensively in the collection class library in Java JDK 1.2. The latest revision of Pizza is designed to exploit this confluence [OW98]. For instance, the user can write code that refers to a class `Collection<A>` and compile this to code that refers to the library class `Collection`. So one can combine the benefits of parametric types with reuse of existing code; this is especially attractive when one realizes that the collection library will be part of all JDK 1.2 compliant browsers, eliminating the need to transmit collection code over the web.

2.2 Virtual types

Here is a roughly equivalent program using virtual types.

```
public class Cell {
  public typedef A as Object;
  protected A value;
  public Cell (A v) { value = v; }
  public void set (A v) { value = v; }
  public A get () { return value; }
}

public class StringCell extends Cell {
  public typedef A as String;
}

public class Test {
  public static void main (String[] args) {
    StringCell c = new StringCell("even");
    c.set("s"+c.get());
    Cell d = c;
    d.set(new Integer(7));  // run-time error
  }
}
```

Here `A` is not a parameter to the class, but instead is treated as a virtual field of the class. Instead of instantiating the parameter to refer to a class `Cell<String>`, one defines a new subclass `StringCell` of `Cell`.

Whereas with parametric types it was a compile-time error to assign a `Cell<String>` to a `Cell<Object>`, with virtual types it is permitted to assign a `StringCell` to a `Cell`. It is still not permitted to place an integer in a string cell, but this is now detected at run-time when the assignment occurs. (This is possible because the new class `StringCell` in effect maintains type information at run-time, and results in behavior that is similar to Java arrays.)

The code above is equivalent to the following code in Java without virtual types.

```
public class Cell {
  public Object cast$A (Object x) { return (Object)x; }
  protected Object value;
  public Cell (Object v) { value = v; }
  public void set (Object v) { value = v; }
  public A get () { return value; }
}

public class StringCell extends Cell {
  public String cast$A (Object x) { return (String)x; }
}

public class Test {
```

```
    public static void main (String[] args) {
      StringCell c = new StringCell("even");
      c.set("s"+(String)c.get());
      Cell d = c;
      d.set(d.cast$A(new Integer(7)));
    }
  }
```

Each virtual type declaration for the type variable `A` is now replaced by code for a method `cast$A` that accepts an object and casts it to the class corresponding to `A`. As before, method calls are translated to add casts where appropriate. But now sometimes calls are required to the `cast$A` methods, and these casts may fail at run-time. In this case, the call to `cast$A` in the last line will fail, since the call dynamically selects the method in `StringCell` but passes it an `Integer` argument.

It does not appear possible to layer virtual types over existing libraries in the same way as with parametric types. For instance, the user could not write code that refers to a class `Collection` with a virtual type and compile this to code that refers to the library class `Collection`.

In summary, parametric types provide more checking at compile-time, while virtual types are more flexible. Parametric types resemble the idiom used in Java to represent polymorphism in a class (such as the collection class library), while virtual types resemble the mechanism used in Java to implement polymorphism over arrays.

3 Strengths of parametric types

Here is something that is easy to do with parametric types, but hard to do with virtual types. The example is given in Pizza, with type parameters written between angle brackets `<...>`.

```
  public class Pair<Fst, Snd> {
    public Fst fst;
    public Snd snd;
  }

  public class List<A> {
    ...
    public <B> List<Pair<A,B>> zip(List<B> yl) { ... }
  }
```

So `zip` is a method where the receiver is a list with elements of type `A` and the argument is a list with elements of type `B` and that returns a list of pairs with components of types `A` and `B`. The types here are so expressive that it is easy to guess what `zip` does: it pairs corresponding elements of two lists. Thus if `xl` is the list of integers `[1,2,3]` and `yl` is the list of strings `["a", "b", "c"]`, then `xl.zip(yl)` returns the list of pairs of integers and strings `[(1,"a"),(2,"b"),(3,"c")]`.

This example is especially compact because both the class `List` and the method `zip` are parameterized (by `<A>` and `<B>` respectively). Some proposals for parametric types (such as Myer, Bank, and Liskov [MBL97] and Agesen, Freund, and Mitchell [AFM97]) allow classes but not methods to be parametric. In this case, the method `zip` can be defined via an inner class with the appropriate type parameters.

```
public class Pair<Fst, Snd> {
  public Fst fst;
  public Snd snd;
}

public class List<A> {
  ...
  public class Zipper<B> {
    public List<Pair<A,B>> zip(List<B> yl) { ... }
  }
}
```

Whereas before one wrote `xl.zip(yl)`, now one would write `xl.new Zipper().zip(yl)`. (Some proposals require explicit types in expressions that create objects, so this would become `xl.new Zipper<B>().zip(yl)` when `yl` has type `List<B>`.) This is more awkward and performs an extra allocation, but permits essentially the same program structure.

It's a frustrating exercise to attempt to express the same information with virtual types. Here is an attempt to do so.

```
public class Pair {
  public typedef Fst as Object;
  public typedef Snd as Object;
  public Fst fst;
  public Snd snd;
}

public class List {
  public typedef A as Object;
  public class Zipper {
    public typedef B as Object;
    public class PairAB extends Pair {
      public typedef Fst as A;
      public typedef Snd as B;
    }
    public class ListB extends List {
      public typedef Elt as B;
    }
    public class ListPairAB extends List {
      public typedef Elt as PairAB;
    }
    public ListPairAB zip (ListB yl) { ... }
  }
```

```
}
```

The proliferation of class definitions is bad enough, but there is a more serious problem: each argument to `zip` must explicitly extend the class `List.Zipper.ListB`. So this solution is awkward and of restricted applicability.

3.1 Collections and subtyping

The following observation is due to Gilad Bracha.

Often one wants one collection class to inherit from another. For instance, the class for lists may inherit from a more general class for collections.

This is easy to arrange with parametric types. Here is the code outline in Pizza.

```
public class Collection<A> { ... }

public class List<A> extends Collection<A> { ... }
```

Now one can pass, say, an argument of type `List<String>` where one of type `Collection<String>` is expected.

Again, this is more problematic with virtual types. The obvious approach does not work.

```
public class Collection {
  public typedef A as Object;
  ...
}

public class List extends Collection { ... }

public class StringCollection extends Collection {
  public typedef A as String;
}

public class StringList extends List {
  public typedef A as String;
}
```

The problem here is that `StringList` does not extend `StringCollection`, so the former cannot be passed where the latter is expected.

One can do a little better by exploiting inner classes.

```
public class Collection {
  public typedef A as Object;
  ...
  public class List extends Collection { ... }
}
```

Here it is essential that `List` be an inner class of `Collection`, so that it inherits use of the virtual type `A`. Now one can declare, say, collections of strings.

```
public class StringCollection extends Collection {
  public typedef A as String
}
```

And now `StringCollection.List` is a subtype of `StringCollection`, as desired. However, this approach requires that the designer of collections has the foresight to include lists. A crucial flexibility of object-oriented languages, that one may define lists long after collections, has been lost.

3.2 Discussion

The difficulties with virtual types appear to arise from two sources. One is sheer length. The other is that virtual types relate solely via subtyping, which in Java must be explicitly declared by the user, whereas the relationship between parametric types is structural. Thus virtual types require one to carefully set up relations between types in advance, whereas with parametric types these relations fall out naturally. Hence, parametric types support program structures that are difficult or impossible to support with virtual types.

4 Strengths of virtual types

The previous examples focussed attention on strengths of parametric types and weaknesses of virtual types. Here is another example, aimed to focus attention on weaknesses of parametric types and strengths of virtual types: families of classes.

Palsberg and Schwartzbach [PS94] tell a compelling story about a common situation where families of classes arise. We will repeat that story here, and then abstract from it to give two examples, 'lists with lengths' and 'alternating lists with lengths'. (The examples are closely related to the 'lists with lengths' and XY-grammar examples of Palsberg and Schwartzbach.)

Consider a processor for a programming language. An early phase of the processor uses abstract syntax trees. These are represented by a family of classes: one class for each non-terminal of the grammar. (Each of these classes might in turn have subclasses to represent each alternative production for the non-terminal.) A later phase of the processor uses annotated abstract syntax trees (they might be annotated with types, flow analysis information, or the like). These are represented by a second family of classes, each original class being subclassed to add the annotation. There might be multiple families, one for each phase.

The simplest possible abstraction of this idea is based on the grammar:

```
X ::= char X | empty
```

which corresponds to lists of characters. The inherited class is augmented with a simple annotation, the length of the list.

The next simplest abstraction is based on the grammar

```
X ::= char Y | empty
Y ::= float X | empty
```

which corresponds to lists that consist alternately of characters and floats. Again, the inherited classes augment each list with its length.

Although these examples are simple, the analogy with grammars and annotations should make it clear that they correspond to a range of examples of interest.

Here is the second example, rendered with virtual types.

```
public class XList {
  public typedef YThis as YList;
  protected char h;
  protected YThis t;
  public XList (char h, YThis t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public YThis tail () { return t; }
  public void setHead (char h) { this.h = h; }
  public void setTail (YThis t) { this.t = t; }
}

public class YList {
  public typedef XThis as XList;
  protected  float h;
  protected XThis t;
  public YList (float h, XThis t) {
    super(); setHead(h); setTail(t);
  }
  public float head () { return h; }
  public XThis tail () { return t; }
  public void setHead (float h) { this.h = h; }
  public void setTail (XThis t) { this.t = t; }
}

public class LenXList extend XList {
  public typedef YThis as LenYList;
  protected int l;
  public LenXList (char h, YThis t) {
    super(h,t);
  }
  public void setTail (YThis t) {
    super.setTail(t);
    if (t == null) l = 1; else l = 1+t.length();
  }
  public int length () { return l; }
}

public class LenYList extend YList {
```

```
    public typedef XThis as LenXList;
    protected int l;
    public LenYList (float h, XThis t) {
      super(h,t);
    }
    public void setTail (XThis t) {
      super.setTail(t);
      if (t == null) l = 1; else l = 1+t.length();
    }
    public int length () { return l; }
  }

  public class Breakit {
    public void breakit () {
      XList xl = new LenXList('a', null);
      YList yl = new YList(3.142f, null);
      xl.setTail(yl);  // run-time error
    }
  }
```

Observe that `XThis` and `YThis`, rather than `XList` and `YList`, are used for the types of the list tails. Thus, within `XList` it is guaranteed is that the tail is a `YList` (and vice versa), while within `LenXList` it is guaranteed that the tail is a `LenYList` (and vice versa). Hence in the `setTail` method in `LenXList`, the new tail of type `YThis` may safely be called with the `length` method (and ditto for `LenYList` with a tail of type `XThis`).

To demonstrate why covariance (and so-called *binary methods*) are difficult, a `breakit` method has been included. In the absence of suitable run-time or compile-time checks, the `breakit` code would violate the invariant that the tail of a list with length is itself a list with length. In Thorup's version of virtual types, the call labeled above signals a run-time error.

It's a frustrating exercise to express the same information with parametric types. Here is an attempt to do so. The translation is presented mainly because it offers some interesting insights, as it is too complex to use in practice.

Each class is paired with a 'functor' class (indicated by the suffix `F`), which takes one parameter for each virtual type. The actual class is derived by instantiating the parameters appropriately.

This translation gives `extends` an (arguably counterintuitive) meaning, in that the extended class is a subclass of the generator, not the named class. This sad complication does carry with it a happy benefit, as it causes the `breakit` method to fail at compile-time rather than run-time: one cannot assign a `LenXList` to an `XList`, since the former no longer extends the latter.

In the following the bodies are omitted since they are identical to the initial example (save that constructor names are changed as appropriate, so that the constructor `XList` becomes `XListF` in that class, and so on). The class `XList` is a fixpoint of the functor `XListF`, and so on. The fixpoint classes only contain constructor definitions, which are required since they cannot be inherited.

```
public class XListF
  <XThis extends XListF<XThis,YThis>,
   YThis extends YListF<YThis,XThis>>
{ ... }

public class YListF
  <YThis extends YListF<YThis,XThis>,
   XThis extends XListF<XThis,YThis>>
{ ... }

public class XList extends XListF<XList,YList> {
  public XList (char h, YList t) { super(h, t); }
}

public class YList extends YListF<YList,XList> {
  public YList (char h, YList t) { super(h, t); }
}

public class LenXListF
  <XThis extends LenXListF<XThis,YThis>,
   YThis extends LenYListF<YThis,XThis>>
  extends XListF<XThis,YThis>
{ ... }

public class LenYListF
  <YThis extends LenYListF<YThis,XThis>,
   XThis extends LenXListF<XThis,YThis>>
  extends YListF<YThis,XThis>
{ ... }

public class LenXList
    extends LenXListF<LenXList,LenYList> {
  public LenXList (char h, LenYList t) { super(h, t); }
}

public class LenYList
    extends LenYListF<LenYList,LenXList> {
  public LenYList (char h, LenXList t) { super(h, t); }
}

public class Breakit {
  public void breakit () {
    XList xl = new LenXList('a', null); // compile-time error
    YList yl = new YList(3.142, null);
    xl.setTail(yl);
  }
}
```

Now the `breakit` method fails at compile time rather than run time. The marked line is a type error because `LenXList` is no longer a subtype of `XList`.

If the above code referred to `this`, an additional trick would be required to give `this` the appropriate type (namely, `XThis` within `XList`, and `YThis` within `YList`). Replace `this` by calls to a method `this.asXThis()`, and add to `XListF` the abstract method declaration

```
abstract XThis asXThis();
```

and then add to `XList` the method body

```
public XList asXThis() { return this; }
```

and similarly for `LenXList`. The types work out because `XThis` is instantiated to `XList` in `XList`.

This translation of virtual types into parameterized types closely resembles the usual semantics of object-oriented features in terms of F-bounded polymorphism [CCHOM89,Bru94].

4.1 Discussion

Mutually recursive families of classes are common in programming. Hierarchically related classes are common in object oriented programming. The combination of the two, hierarchies of mutually recursive classes, is less common but occasionally of central importance. Palsberg and Schwartzbach give one compelling example of this sort, a grammar (one mutually recursive family) and an annotated grammar (a second mutually recursive family, hierarchically related to the first). Here we have given a simplified example to capture the essence of the problem, alternating lists (one family) and alternating lists with lengths (a second family, hierarchically related to the first). Parameterized types offer no special support for hierarchical recursive families, but virtual types cater to them neatly.

The `MyType` or `ThisType` construct evolved to deal with the special case of hierarchical families where each family member only needs to refer recursively to itself. Lists are a classic example of a recursive class, and adding lists with lengths gives a classic example for which `ThisType` is perfectly suited. Generalizing to alternating lists with lengths would take one beyond the scope to which `ThisType` applies, but where virtual classes do well. On the other hand, `ThisType` is subject to static checking, while virtual types traditionally require dynamic checking. In the next section, we suggest a generalization to `ThisType` as a way to resolve this conflict.

5 Providing virtual types statically

In this section of the paper we present a proposal for extending Java (or indeed any statically typed object-oriented language) with a construct which is similar to virtual types, but which can be statically type-checked with rules which can be easily understood by programmers. In particular we show that if $I_1, \ldots, I_n$ is a system of interfaces whose definitions are mutually referential (i.e., each may refer to any of the others), then it is possible to define (in a type-safe way) a system of subinterfaces, $J_1, \ldots, J_n$ of the I_i such that all inherited references to the I_i's behave as references to J_i's in the subinterfaces. Moreover classes implementing these interfaces will also have the signatures of methods and fields change appropriately in subclasses implementing the subinterfaces. An important feature of this construction is that classes defined in this way can be statically type-checked to ensure that no type errors can occur at run-time.

5.1 The simple case: Understanding `ThisType`

In earlier work [BSvG95] the authors solved a similar problem with *self*-referential interfaces and classes using a `MyType` or `ThisType` construct. In an interface or class definition, `ThisType` stands for an interface of `this`, the object being declared or defined. In the most recent languages designed in this research project [BFP97,Bru97], “exact” types were added in order to simplify dealing with binary methods.

The prefix `@` associated with a type indicates an “exact” type. That is, if a variable `aExact` is declared to have type `@A`, then values stored in `aExact` must be of exactly that type, not from an extension. If `B` extends `A`, variable `a` is declared to have type `A`, and `b` has (static) type `B`, then the assignment `a = b` is legal because a variable of type `A` can hold values from any type extending it. However if `aExact` is declared to have type `@A`, then the assignment `aExact = b` will be determined to be illegal by the static type checker since only values of exactly type `A` can be held in `aExact`. In fact, even `aExact = a` is considered illegal since `a` could hold a value of type `B`. Of course one can always insert an explicit cast `aExact = (@A)a` to generate a run-time check and let it pass the type checker.

The ability to specify exact types is helpful in defining homogeneous data structures, but a very important use of exact types is in supporting the use of *binary methods* [BCCHLP95], *i.e.*, methods with a parameter of type `ThisType`, as we shall see below. See below and either [Bru97] or [BFP97] for more detailed explanations of the use of exact types with *binary methods*.

The following example illustrates the use of exact types and `ThisType` in the context of a list whose head is a character and whose tail is a List.

```
interface ListIfc {
  public char head ();
  public @ThisType tail ();
  public void setHead (char h);
  public void setTail (@ThisType t);
}
```

Uses of `ThisType` in method signatures denote an interface which is the same as that being defined. Thus it supports recursive references to the interface. We discuss below how the use of `ThisType` differs from the use of the explicit interface name being defined.

```
public class List implements @ListIfc {
  protected char h;
  protected @ThisType t;
  public List (char h, @ThisType t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @ThisType tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@ThisType t) { this.t=t; }
}
```

Uses of `ThisType` in a method of a class refer to the interface formed by the public methods of the object executing that method. Alternatively, we can say that `ThisType` is exactly the public interface of `this`.

As a first approximation, one might think of `ThisType` as another name for `ListIfc`. However its meaning is a bit more subtle. Just as the meaning of `this` changes when methods are inherited in subclasses, the meaning of `ThisType`, which represents its interface, changes in tandem. Because the meanings of both `this` and `ThisType` change in extensions, all methods in `List` are type-checked under the assumption that `this` has interface `@ThisType` and that `ThisType` extends `ListIfc`.

`List` implements `@ListIfc` because the public methods of `List` are exactly those specified by `ListIfc`. The instance variable `t` of `List` is declared to be exactly `ThisType` in order to assure that the tail of the list has exactly the same interface as the whole list. We shall see below that this is useful in maintaining the consistency of objects defined in extensions. As might be expected, the use of `@ThisType` for the instance variable type leads to the use of the same type as the parameter of `setTail` and the return type of `tail`.

The real advantage in using `ThisType` comes when we define extensions of a class. We can define an extension of `List` which keeps track of the length of the list.

```
public interface LenListIfc extends ListIfc {
  public int length();
}

public class LenList extends List implements @LenListIfc {
  protected int l;
  public LenList (char h, ThisType t) {
    super(h, t);
  }
  public void setTail (@ThisType t) {
    super.setTail(t);
    if (t == null) l = 1; else l = 1+t.length();
  }
  public int length() { return l; }
}
```

Because the instance variable `t` is declared to have type `@ThisType` (rather than `List` or `ListIfc`), when it is inherited its meaning changes along with the meaning of `this`. Thus an object generated by `List` will have a protected instance variable `t` holding a value with interface `@ListIfc`, while the corresponding `t` in an object generated by `LenList` will hold a value with interface `@LenListIfc`. Similarly, the value returned by sending the `tail()` message to a `List` object will be of type `@ListIfc`, while the result returned will be of type `@LenListIfc` if the same message is sent to a value generated from `LenList`. In particular, note that the expression `t.length()` in the body of the `setTail` method of `LenList` type checks correctly only because the type of parameter `t` was declared to be `ThisType`. A type error would have been generated if its type had instead been declared to be `ListIfc`. Thus the use of `ThisType` provides greater flexibility when defining classes with fields or methods whose signatures should be the same as the object.

There is one restriction when using methods with parameters whose type involves `ThisType` (the so-called binary methods), like `setTail`. It is that the corresponding messages can only be sent to objects for which we know the exact types. Thus if `list` has type `@ListIfc` (or `@List`), then `list.setTail(otherList)` will be well-typed only if `otherList` has type `@ListIfc`.

Suppose instead that all we know about `list` is that it has type `ListIfc`. Then `list` could have been generated by `List` or `LenList` (or indeed any other classes which implement the interface). But now if we consider `list.setTail(otherList)` we cannot determine statically what the type of `otherList` should be. If at run-time the value of `list` really comes from `List`, then `otherList` should be of type `@ListIfc`. But if the value of `list` comes from `LenList`, then `otherList` must be of type `@ListIfc`. Note that a run-time error will result in the latter case if `otherList` is from `List`, as it does not have a method corresponding to the `length` message called in the body of `LenList`'s `setTail` method.

There are two cases for type-checking message sends depending on whether or not we know the exact interface of the receiver. If the interface of the receiver is `@T` then all occurrences of `ThisType` in the method signature are replaced by `T`. If we are not provided with the exact interface, but only know the interface of the receiver is `T` then only non-binary messages can be sent and for those messages all occurrences of `ThisType` and `@ThisType` in the signature are replaced by `T`.

Aside from this rule, type checking of most other constructs is straightforward. The only subtlety is that when type checking a class, we presume that `this` has an anonymous class which is an extension of the class being defined, and which implements exactly `ThisType`. For example, an occurrence of `this` in class `List` is type-checked under the assumption that it is from an anonymous class which extends `List` and implements `@ThisType`. `ThisType` is also assumed to be an extension of `ListIfc`.

Type checking is done this way in order to ensure that methods remain type safe in all possible extensions of the class being defined. In particular, `this` has access to all of the instance variables and methods of the class being defined, but may not be restricted to only represent an object of the class being defined. (See [BFP97] or [Bru97] for details.) With these type-checking rules, the language with exact types and `ThisType` supports a static type discipline that guarantees that type errors will not occur at run-time.

The reader may wonder why we have chosen to have `ThisType` denote an interface rather than a class. Suppose instead that we had included a `ThisClass` construct. Here is a reworking of the previous example using `ThisClass`.

```
public interface ListClassIfc {
  public char head ();
  public @ThisClass tail ();
  public void setHead (char h);
  public void setTail (@ThisClass t);
}

public class ListClass implements ListClassIfc {
  protected char h;
  protected @ThisClass t;
  public List (char h, @ThisClass t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @ThisClass tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@ThisClass t) { this.t=t; }
}
```

The use of `ThisClass` greatly reduces flexibility, since we can no longer utilize a variable with the interface type `ListClassIfc` with methods whose signatures mention `ThisClass`. Indeed, suppose `xl` has type `ListClassIfc`, and consider the expression `xl.setTail(yl)`. What type should the parameter `yl` have? Intuitively it should be generated by the same class as `xl`, but we cannot determine statically what class generated `xl`. Hence we cannot statically type check this construct. Because of these problems and our desire to encourage the use of interfaces, we choose for `ThisType` to range over interfaces rather than classes.

5.2 Generalizing `ThisType` to mutually recursive interfaces and classes

We wish to provide a type-safe statically checkable language construct to define mutually referential classes, in which the classes (and their interfaces) are expected to change in parallel in extensions. Since they refer to each other, we need some way of grouping them together and providing the same sort of flexible names as `ThisType`. In particular, we need a way of associating these flexible names with the appropriate classes.

We explicitly associate a `ThisType`-style name to an interface by including it in parentheses immediately after the interface name.

```
interface AnInterface (TType) { ... }
```

Here `AnInterface` is the name of the interface, while `TType` is a type variable used analogously to the `ThisType` interface, which automatically changes in extensions of the interface. We will designate the name following the keyword "`interface`" as the *specific* name of the interface and the parenthesized name as its *variable* name.

We can rewrite interface `ListIfc` and class `List` above using this new notation:

```
public interface ListIfc (TType) {
  public char head ();
  public @TType tail ();
  public void setHead (char h);
  public void setTail (@TType t);
}

public class List (TType) implements @ListIfc {
  protected char h;
  protected @TType t;
  public List (char h, @TType t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @TType tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@TType t) { this.t=t; }
}
```

The parenthesized expression which introduces the variable name `TType` in the declaration of the interface `ListIfc` indicates that the variable name `TType` is to be used as the interface of `this` in classes which implement that interface. Thus the only changes made to the class `List` were to explicitly include the *variable* name of the interface in parentheses and to replace all occurrences of `ThisType` with the variable name `TType`.

Mutually referential interfaces and classes which need to refer to each other's `ThisType` will be grouped together so as to delimit the scope of the definitions. Fortunately, Java 1.1 now includes "inner" or nested interfaces and classes which provide the framework for this grouping. (Thanks to Kresten Thorup for suggesting this use of inner classes.)

The following is a reworking of the alternating list example in the new notation.

```
public interface AltListGrpIfc {
  public interface XListIfc (XThis) {
    char head ();
    @YThis tail ();
    void setHead (char h);
    void setTail (@YThis t);
```

```
  }

  public interface YListIfc (YThis) {
    float head ();
    @XThis tail ();
    void setHead (float h);
    void setTail (@XThis t);
  }
}
```

The intention is that any implementation of the "outer" interface, `AltListGrpIfc` must provide implementations of the "inner" interfaces, `XListIfc` and `YListIfc`. The scope of the variable names of the inner interfaces, `XThis` and `YThis`, includes the entire body of the outer interface, `AltListGrpIfc`.

```
public class AltListGrp implements AltListGrpIfc {
  public static class XList (XThis) implements @XListIfc
  {
    protected char h;
    protected @YThis t;
    public XList (char h, @YThis t) {
      super(); setHead(h); setTail(t); }
    public char head () { return h; }
    public @YThis tail () { return t; }
    public void setHead (char h) { this.h=h; }
    public void setTail (@YThis t) { this.t=t; }
  }

  public static class YList (YThis) implements @YListIfc
  {
    protected float h;
    protected @XThis t;
    public YList (float h, @XThis t) {
      super(); setHead(h); setTail(t);
    }
    public float head () { return h; }
    public @XThis tail () { return t; }
    public void setHead (float h) { this.h=h; }
    public void setTail (@XThis t) { this.t=t; }
  }
}
```

We also make a minor extension to Java, in that we assume that since `AltListGrp` implements `AltListGrpIfc`, the names `XListIfc` and `YListIfc` can be used without qualification inside `AltListGrp`. (Java proper requires one to write `AltListGrp.XListIfc` and `AltListGrp.YListIfc` instead.)

Type checking of the classes is performed similarly to that of the simpler case discussed in the previous section. Inside `XList`, `this` is presumed to have an anonymous class which is an extension of `XList` and which implements `@XThis`, while inside `YList`, similar assumptions are made of `this` and `@YThis`. Because each class and interface may be replaced by extensions in a subclass of `AltListGrp`, we only assume that `XThis` extends `XListIfc` and `YThis` extends `YListIfc` when type-checking the classes. We emphasize that the constraints on both `XThis` and `YThis` are available when type-checking each of `XList` and `YList`.

The notion of binary methods must be extended for this notation. A method of an inner interface is considered "binary" if any of its parameters have a type which is a variable name of any other inner interface of the same outer interface. Thus the methods named `setTail` in `XListIfc` and `YListIfc` are binary because they have parameters of type `YThis` or `XThis`. For the same reasons as in section 5.1, binary messages may only be sent to objects for which we know the exact types.

We can now define the subinterfaces and subclasses.

```
public interface LenAltListGrpIfc extends AltListGrpIfc
{
  public interface LenXListIfc (XThis) extends XListIfc
  {
    public int length();
  }

  public interface LenYListIfc (YThis) extends YListIfc
  {
    public int length();
  }
}

public class LenAltListGrp extends AltListGp
{
  public static class LenXList (XThis)
            extends XList implements @LenXListIfc
  {
    protected int l;
    public LenXList (char h, @YThis t) {
      super(h, t);
    }
    public void setTail (@YThis t) {
      super.setTail(t);
      if (t == null) l = 1; else l = 1+t.length();
    }
    public int length () { return l; }
  }

  public static class LenYList (YThis)
            extends YList implements @LenYListIfc
  {
    protected int l;
```

```
      public LenYList (float h, @XThis t) {
        super(h,t);
      }
      public void setTail (@XThis t) {
        super.setTail(t);
        if (t == null) l = 1; else l = 1+t.length();
      }
      public int length () { return l; }
    }
  }
```

The subinterface `LenAltListGrpIfc` contains extensions of the interfaces `XListIfc` and `YListIfc`, while the subclass `LenAltListGrp` contains extensions of the classes `XList` and `YList` from `AltListGrp`. This time both classes are type checked assuming that `XThis` extends `LenXListIfc` and `YThis` extends `LenYListIfc`. If a new interface redefining a variable interface name is not provided in an extension of the outer class, the old one is inherited unchanged. For example, if an interface with `XThis` as its variable name were not included in `LenAltListGrpIfc`, then uses of `XThis` inside the "outer interface" would be interpreted as being tied to `XListIfc`.

We access the inner classes and interfaces by qualifying their names with the enclosing class.

```
  public class Useit {
    public void useit () {
      @AltListGrpIfc.XThis xl
        = new AltListGrp.XList('a',null);
      @AltListGrpIfc.YThis yl
        = new AltListGrp.YList('b',null);
      xl.setTail(yl);
    }
  }
```

When using qualified names, we can choose to use either the specific or variable name of the interface. For maximum flexibility (and in anticipation of the next section) we have chosen to use the latter (*e.g.*, `XThis` rather than `XListIfc`) in this example.

We expect it will be more useful to use variable names (like `YThis`) for interfaces, rather than the specific names (like `YListIfc`). However, specific names of classes must be used in `new` expressions, since classes must be used to create new instances. There may also be circumstances in which the specific names of interfaces are used because the programmer does not wish an interface specification to change in extensions. Thus if we wished the method `setTail` to take a parameter of type `YListIfc` in all subclasses (rather than having it change with the subclass) then we could write `YListIfc` rather that `YThis` in its declaration.

The example in Appendix A also includes a "`use` clause" which can be used to bring externally-defined interfaces and classes into a collection so that extensions can also depend on extensions of those declarations. While this will likely not be used regularly, there are times when it is very helpful to bring in externally defined interfaces or classes.

6 Parametric polymorphism with mutually recursive groups

These new constructs interact smoothly with parametric polymorphism, both in defining polymorphic inner classes, and in using the "outer" interface as a constraint for type variables. As an example of the first sort we could change our implementation of `AltListGrp` to hold values of a type specified by a type parameter.

```
public interface PolyAltListGrpIfc
  <XType extends Object, YType extends Object>
{
  public interface XListIfc (XThis) {
    public XType head ();
    public @YThis tail ();
    public void setHead (XType h);
    public void setTail (@YThis t);
  }

  public interface YListIfc (YThis) {
    public YType head ();
    public @XThis tail ();
    public void setHead (YType h);
    public void setTail (@XThis t);
  }
}

public class PolyAltListGrp <XType extends Object, YType extends Object>
  implements PolyAltListGrpIfc<XType,YType>
{
  public static class XList (XThis) implements XListIfc
  {
    protected XType h;
    protected YThis t;
    ...
  }

  public static class YList (YThis) implements YListIfc
  {
    protected YType h;
    protected XThis t;
    ...
  }
}
```

The class `AltListGrp.XList` defined earlier is equivalent to `PolyAltListGrp<char,float>.XList`, and similarly for `YList`.

A different use of polymorphism arises when we use an "outer" interface as a type constraint.

```
public class Useit<T extends AltListGrpIfc> {
  ... T.XThis ... T.YThis ...
}
```

Because the type variable `T` extends `AltListGrpIfc`, it must support inner interfaces with variable names `XThis` and `YThis`. These inner interfaces are mutually referential and extend `AltListGrpIfc.XThis` and `AltListGrpIfc.YThis`, respectively. For example, an expression with type `@T.XThis` can be sent a `setTail` message with a parameter of type `@T.YThis`. As a result, groups of interfaces (packaged as inner interfaces) can be passed around and used polymorphically. This extension would be simple to accommodate in proposals where parametric types are implemented by expansion into specialized instances, as in the proposal of Agesen *et.al.* [AFM97] or in the heterogeneous translation of Pizza [OW97]. It is not clear how to implement the extension in a homogeneous translation where type parameters are erased.

7 Conclusion

Parametric types and virtual types each do things well that the other does poorly. Both parametric types and virtual types have been given semantics expressed as translations from extensions to Java into Java as it stands (see [Tho97,OW97,OW98]), and these translations look remarkably similar.

One way forward is to tack features that mimic parametric types onto virtual types (as proposed by Cartwright and Steele [CS97] and Thorup and Torgersen [TT98]), or to tack features that mimic virtual types onto parametric types as proposed here.

Our proposal supports parametric types directly, and supports virtual types by using inner interfaces and classes to group together mutually recursive declarations that may be changed simultaneously. An important advantage of this language design is that all type checking can be done statically — it does not require extra dynamic type checking, and the resulting type system guarantees type safety.

A statically safe method to model virtual types was proposed by Torgersen [Tor98]. His work is similar to, but independent from ours. Where we write `interface I (IThis)`, Torgersen would write:

```
public interface I{
  IThis <= I;
```

Instead of relying on exact types, Torgersen requires that a class containing a virtual type is `final bound` (so it cannot be redeclared in subclasses) before it can be used as an ordinary type:

```
public interface IFinal extends I{
  IThis = IFinal;
```

The meaning of `IThis` is thus fixed in all extensions of `IFinal`. Only those classes in which all virtual types are final bound can have instances. If the only type known for an object in a context contains non-final bound virtual types then no binary messages can be sent to it.

Torgersen's scheme has the advantage of needing less machinery, but some inheritance hierarchies cannot be expressed using his technique. Moreover, one must often create new extensions of classes and interfaces just to finalize a virtual type.

Thorup and Torgersen have suggested that virtual types can take on more of the benefits of parametric types if structural subtyping is used rather than declared subtyping, but the details of their proposal remain to be worked out [TT98].

It is interesting to note that many of the difficulties encountered here have to do with providing explicit types for code which clearly will execute without errors. Languages with type inference shift this work from the programmer to the inference system. Thus in Objective ML [RV98] one can write the code in a way similar to that given here and have the system deduce safe typings. The trade-off is that in such systems it is harder to see what changes will be allowable in extensions since the type information is not explicit .

Our notation can be considered as a variant of the proposal for handling inheritance in the presence of mutual recursion put forward by Palsberg and Schwartzbach [PS94]. It differs from their proposal by requiring the interaction of inheritance with recursion to be explicitly declared, which is interesting because they claimed such an option was not possible.

We have presented this proposal as an extension of Java (in particular an extension of the earlier proposal [Bru97]), but it could relatively easily be adapted to other statically-typed object-oriented languages.

Does our proposal constitute the ultimate solution to integrating parametric and virtual types? We think we've made a useful step, but further study is required.

References

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

[BCCHLP95] 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-Oriented 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.

[Bru94] Kim B.Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Bru97] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997. Available via <http://www.cs.williams.edu/~kim/README.html>.

[Bru97b] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *European Conference on Object-Oriented Programming*, pages 27–51, LNCS 952, Springer-Verlag, 1995.

[CCHOM89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell, F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, ACM, 1989.

[CS97] Corky Cartwright and Guy Steele. Yet another parametric types proposal. Message to Java genericity mailing list, August, 1997.

[GRJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[KM96] Andrew Koenig and Barbara Moo. *Ruminations on C++.* Addison-Wesley, 1996.

[KMMN83] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Abstraction mechanisms in the Beta programming language. In *Symposium on Principles of Programming Languages*, ACM, 1983.

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

[MM89] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Object-Oriented Programming: Systems, Languages, and Applications*, ACM, 1989.

[MMN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

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

[OW98] Martin Odersky and Philip Wadler. Leftover Curry and reheated Pizza: How functional programming nourishes software reuse. In *IEEE Fifth International Conference on Software Reuse*, Vancouver, BC, June 1998.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* Wiley, 1994.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object-Oriented Systems*, 4(1): 27–50, 1998.

[Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.

[Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513–523. North-Holland, Amsterdam.

[Str67] C. Strachey, Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, pages 444–471, LNCS 1241, Springer-Verlag, 1997.

[Tor98] Mads Torgersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

[TT98] Kresten Krab Thorup and Mads Torgersen. Structural virtual types. Informal session on types for Java, *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

A The subject-observer example

In this section we examine another example of the use of virtual types and demonstrate how this would be written in the Java extension suggested in the paper. Thorup's paper [Tho97] used the following example based on the Observer pattern from [GRJV94].

```
public class Observer {
  public typedef SType as Subject
  public typedef EventType as Object;
  public void notify (SType s, EventType e) { ... }
}

public class Subject {
  public typedef OType as Observer;
  public typedef EventType as Object;
  protected OType observers[];
  ...
  public void notifyObservers(EventType e) {
    int n = observers.length;
    for (int i = 0; i < n; i++)
      observers[i].notify(this,e);
    }
  }

public class WindowObserver extends Observer {
  public typedef SType as WindowSubject;
  public typedef EventType as WindowEvent;
}

public class WindowSubject extends Subject {
  public typedef OType as WindowObserver;
  public typedef EventType as WindowEvent;
  ...
}
```

The following is a reworking of Thorup's example in our notation.

```
public interface SubObsGrpIfc{
  public interface ObserverIfc (OType) {
```

```
    void notify (SType s, EventType e);
  }

  public interface SubjectIfc (SType) {
    ...
    void notifyObservers (EventType e);
  }
  public use Object (EventType);
}

public class SubObsGrp implements SubObsGrpIfc {
  public static class Observer (OType) implements @ObserverIfc
  {
    public void notify (SType s, EventType e) { ... }
  }

  public static class Subject (SType) implements @SubjectIfc
  {
    protected @ThisObserver observers[];
    ...
    public void notifyObservers(EventType e){
      int n = observers.length;
      for (int i = 0; i < n; i++)
        observers[i].notify(this,e);
    }
  }
}
```

The only feature not previously introduced is the "`use`" clause in the definition of `SubObsGrpIfc`. This indicates that the variable type `EventType` will be allowed to vary in extensions of `SubObsGrpIfc`, and hence that methods of `SubObsGrp` should be type checked under only the assumption that `EventType` extends `Object`.

By our earlier remarks, the occurrence of `this` in the body of `notifyObservers` is type-checked as having an anonymous class which extends `Subject` and which implements `@ThisSubject`.

```
public interface WindowSubObsGrpIfc extends SubObsGrpIfc
{
  public interface WindowObserverIfc (OType) extends ObserverIfc
  { ... }

  public interface WindowSubjectIfc (SType) extends SubjectIfc
  { ... }

  public use WindowEvent (EventType);
}

public class WindowSubObsGrp extends SubObsGrp
  implements WindowSubObsGrpIfc
```

```
{
  public static class WindowObserver (OType)
    extends Observer implements WindowObserverIfc
  { ... }

  public static class WindowSubject (SType)
    extends Subject implements WindowSubjectIfc
  { ... }
}
```

The "`use`" clause in `WindowSubObsGrpIfc` introduces `WindowEvent` as the new interpretation for `EventType`. Type-checking rules require that the new interpretation of `EventType` be an extension of the meaning in the super-interface, in this case `Object`. As with the inner interfaces, if no replacement is provided for a variable name specified in a "`use`" clause, the old one is inherited.