A statically safe alternative to virtual types
Extended Abstract

Kim B. Bruce*
Williams College
kim@cs.williams.edu

Martin Odersky
University of South Australia
odersky@cis.unisa.edu.au

Philip Wadler
Bell Labs, Lucent Technologies
wadler@research.bell-labs.com

Abstract

Parametric types and virtual types have recently been proposed as extensions to Java to support genericity. In this paper we examine both in order to 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 have been proposed by Myers, Bank, and Liskov [MBL97], Agesen, Freund, and Mitchell [AFM97], by Bruce [Bru97], and by Odersky and Wadler as part of Pizza [OW97]. Virtual types have been proposed 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 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, and Kresten Thorup made particularly significant contributions.

The rest of the paper is organized as follows. Section 1 presents examples which are easy to express with parametric types, but are hard to express with virtual types. Section 2 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.

In Section 3 we propose a notation which is similar to virtual types, but which can be statically type checked. Section 4 shows how this can be combined with parametric types in a smooth way. Section 5 summarizes our results. Appendices provide another example of the new notation and provide more details on particular features of the language.

2 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;
}
```

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

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

The types here are so expressive that it's easy to guess what zip does: pair corresponding elements of two lists. Thus if `xl` is a list of integers (say `[1,2,3]`) and `yl` is a list of strings (say `["a", "b", "c"]`), then `xl.zip(yl)` is a list of pairs of integers and strings (say `[(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 {
  typedef Fst as Object;
  typedef Snd as Object;
  public Fst fst; public Snd snd;
}

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

2.1 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 {
  typedef A as Object;
  ...
}

public class List extends Collection { ... }

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

public class StringList extends List {
  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 {
  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{
  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 Strengths of virtual types

The previous examples focussed attention on strengths of parametric types and weaknesses of virtual types. Here is a second 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 {
  deftype 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 {
  deftype 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 {
  deftype 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 {
  deftype 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);  // fails at run-time
  }
}
```

Observe that `XThis` and `YThis`, rather than `XList` and `YList`, are used for the types of the list tails. 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 attempt 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> {
  XList (char h, YList t) { super(h, t); }
}

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

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

public class Breakit {
  public void breakit () {
    XList xl = new LenXList('a', null); // type 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 `XList$gen` 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 parameterised types closely resembles the usual semantics of object-oriented features in terms of F-bounded polymorphism [CCHOM89, Bru94].

4 Providing virtual types statically

In this section of the paper we present a proposal for extending Java 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.

4.1 The simple case: Understanding `ThisType`

In earlier work [BSvG95, BFP97, Bru97] 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 the object being declared or defined. The following example illustrates the use of `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);
}

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; }
}
```

Here, within `ListIfc` the type variable `ThisType` stands for the type of `this` for any class that implements the interface `ListIfc`, and within `List` the type variable `ThisType` stands for the type of `this` for any class that implements the public interface of `List`, which is exactly `ListIfc`.

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 in the move to subclasses, the meaning of `ThisType`, which represents the interface of `this`, changes as well. Thus in `List`, all methods are type-checked under the assumption that `this` has interface `@ThisType` and that `ThisType` extends `ListIfc`.

The prefix `@` indicates an "exact" type. That is, if a variable `a_exact` has type `@A`, then values stored in `a_exact` must be of exactly that type, not from an extension. Thus if `B` extends `A`, `a` has type `A`, and `b` has 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 `a_exact` has type `@A`, then the assignment `a_exact = b` will be illegal since only values of exactly type `A` can be held in `a_exact`. Exact types are needed in handling *binary methods* [BCC+95], *i.e.*, methods with a parameter of type `ThisType`. See Appendix A and either [Bru97] or [BFP97] for more detailed explanations of the use of exact types with *binary methods*.

The 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 {
  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` has 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 of type `@ListIfc`, while the corresponding `t` in an object generated by `LenList` will hold a value of type `@LenListIfc`. Similarly the value returned by sending the `tail()` message to a value generated by `List` 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`.

Thus the use of `ThisType` provides greater flexibility when defining classes with fields or methods whose signatures should be the same as the object.

We have chosen to have `ThisType` designate an interface rather than a class. Suppose instead that we had included a `ThisClass` construct. Here is a reworking of the previous example using this construct.

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

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; }
}
```

This greatly reduces flexibility, since we cannot utilize a variable with the interface type `ListClassIfc`. 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 tell statically what class generated `xl`. Hence we cannot statically type check this construct. For this reason, we choose for `ThisType` to range over interfaces rather than classes.

4.2 Generalizing `ThisType` to mutually recursive interfaces and classes

We wish to provide a statically checkable language construct to define mutually referential classes, where 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.

The only syntactic extensions to Java that we need are two new keywords, `as` and `use`, as well as clauses which use these new keywords to associate flexible names for classes. We associate a `ThisType`-style name to an interface with the following notation.

```
interface Ifc as TT { ... }
```

Here `Ifc` is the name of the interface, while `TT` 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 name following the keyword "`as`" as its *variable* name.

We warn the reader that we have interchanged the order of the variable and specific names in the "`as`" clause from that used in the virtual types proposal. The concrete syntax is still under discussion and may change in the future.

We can rewrite class `List` above using the "`as`" construct.

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

public class List as 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 "`as`" clause which introduces the variable name `TType` in the declaration of the interface `ListIfc` indicates that `TType` is to be used as the interface of `this` in classes which implement that interface. The only changes made to the class `List` were to include the `implements` clause associating it with the interface and to replace all occurrences of the `ThisType` with the variable name `TType` from `ListIfc`.

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

public class AltListGrp implements AltListGrpIfc {
  public static class XList as 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 as 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; }
  }
}
```

The intention is that any implementation of the "outer" interface, `AltListGrpIfc` must provide implementations of the "inner" interfaces, `XListIfc` and `YListIfc`. Because the "outer" interface `AltListGrpIfc` delimits the scope of the `XThis` and `YThis` type variables, we do not need to include a definition of `YThis` in `XListIfc` or of `XThis` in `YListIfc`.

We also make a minor extension to Java, in that we assume that since `AltListGrp` implements `AltListGrpIfc` that 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 done as follows. Inside `XList`, `this` has type `@XThis`, while inside `YList`, `this` has type `@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`.

We can now define the subinterfaces and subclasses.

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

public class LenAltListGp
  extends AltListGp
{
  public static class LenXList as 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 as 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 default is that that the old one is inherited unchanged. For example, if an interface naming `XThis` in its "`as`" clause were not included in `LenAltListGrpIfc`, then uses of `XThis` inside the "outer inteface" 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.

The example in Appendix B 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 items.

It will likely be more useful to use variable names (like `YThis`), rather than the specific names (like `YList`). Specific names must be used in `new` expressions, since classes but not interfaces can create new instances. There may also be circumstances in which the specific names 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 the `YListIfc` rather that `YThis` in its declaration.

5 Parametric polymorphism with mutually recursive groups

These constructs interact smoothly with parametric polymorphism, both in defining polymorphic inner classes, and in using the "outer" class as a constraint for type variables. For example 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 as XThis {
    public XType head ();
    public @YThis tail ();
    public void setHead (XType h);
    public void setTail (@YThis t);
  }
  public interface YListIfc as 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 as XThis
    implements XListIfc
  {
    protected XType h;
    protected YThis t;
    ...
  }
  public static class YList as YThis
    implements YListIfc
  {
    protected YType h;
    protected XThis t;
    ...
  }
}
```

The previous `AltListGrp.XList` 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 specialised 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.

6 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]), and these translations look remarkably similar.

While an extension of Java supporting parametric types can be statically typed (see [OW97, Bru97, AFM97, MBL97], for example), Thorup's [Tho97] proposed extension supporting virtual types requires dynamic type checking to preserve type safety. One can translate virtual types into parametric types in a typesafe way, as shown in Section 3, but the result is too cumbersome to be useful in practice.

One way forward is to tack features that mimic parametric types onto virtual types (as proposed by Cartwright and Steele [CS97]), or to tack features that mimic virtual types onto parametric types as proposed here in section 3.

In sections 4 and 5 we present a proposal for such an extension. It 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 is that all type checking can be done statically – it does not require extra dynamic type checking.

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 as IThis`, Torgersen would declare `I` as an inner interface in a wrapper class which also contains the virtual type declaration `typedef IThis as I`. This is strikingly similar to our treatment of mutually recursive classes. Instead of relying on exact types, Torgersen requires that a class containing a virtual type is declared `final` (so it cannot be redeclared in subclasses) before it can be used as an ordinary type. His scheme has the advantage of needing less machinery, but some inheritance hierarchies cannot be expressed using his technique.

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.

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.

[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-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-Verlang, 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.

[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.

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

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[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.

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

A Exact Types

Exact types are necessary when sending "binary" messages to objects. A binary method is one which has a parameter of type `ThisType` in the Java extension proposed in [Bru97]. The method `setTail` in `List` or `ListIfc` is an example of a binary method. The receiver of a binary message (*i.e.*, a message corresponding to a binary method) must be an exact type.

Suppose not. Let `ExtendedList` extend `List` by adding a new public method `doSomething`, which is not included in `List`, and by redefining the method `setTail` in such a way that it includes sending the message `doSomething` to its argument, `tail`. This will not be a problem since `ExtendedList` is type-checked under the assumption that `ThisType` includes all public methods of `ExtendedList`.

Now let `xl` and `yl` be variables declared to have type `ListIfc`. Thus the values of `xl` and `yl` at run time might have been generated by `List`, `ExtendedList`, or any other class implementing `ListIfc`. Is `xl.setTail(yl)` well-typed?

If `xl` and `yl` both hold values generated by the same class then there will be no problem. However, suppose `xl` holds a value generated from `ExtendedList` and `yl` a value from `List`. Then the version of `setTail` in `ExtendedList` will send the message `doSomething` to `yl` which is not capable of responding to the message. Thus `xl.setTail(yl)` should not be well-typed.

However, if both `xl` and `yl` have type `@ListIfc` then `xl.setTail(yl)` will always be type safe as the code for `setTail` in any class implementing `@ListIfc` will be type checked under the assumption that `ThisType` extends `ListIfc`, so it cannot depend on any methods not contained in `ListIfc`.

The notion of binary methods must be extended for the notation proposed in section 4. In this case, 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`) is binary because it has a parameter of type `YThis` (respectively, `XThis`).

Thus the call of `setTail` below will not statically type check.

```
public class Breakit {
  public void breakit () {
    AltListGpIfc.XThis xl
      = new LenAltListGrp.LenXList('a', null);
    AltListGpIfc.YList yl
      = new LenAltListGrp.LenYList(3.142f, null);
    xl.setTail(yl);  // type error
  }
}
```

However, the following code is type correct since the type of the receiver is an exact type.

```
public class Useit {
  public void useit () {
    @AltListGpIfc.XThis xl
      = new AltListGroup.XList('a', null);
    @AltListGpIfc.YThis yl
      = new AltListGroup.YList(3.142f, null);
    xl.setTail(yl);  // now type correct
  }
}
```

Because of space limitations we are unable to include all of the type checking rules for this extension, but the reader is referred to [BFP97] for the typing rules for a similar language, but in which the conventions for exact and "flexible" types are reversed.

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

public class Subject {
  typedef OType as Observer;
  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 {
  typedef SType as WindowSubject;
  typedef EventType as WindowEvent;
}

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

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

```
public interface SubObsGrpIfc{
  public interface ObserverIfc as OType {
    void notify (SType s, EventType e);
  }
  public interface SubjectIfc as SType {
    ...
    void notifyObservers (EventType e);
  }
  public use Object as EventType;
}

public class SubObsGrp implements SubObsGrpIfc {
  public static class Observer as OType
    implements @ObserverIfc
  {
    public void notify (SType s, EventType e) {...}
  }
  public static class Subject as 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 type `EventType` will be allowed to vary in extensions of `SubObsGrpIfc`, and hence that methods of `SubObsGrp` should be type checked under only under the assumption that `EventType` extends `Object`.

By our earlier remarks, the occurrence of `this` in the body of `notifyObservers` is type-checked with type `@ThisSubject`.

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

```
    public interface WindowSubjectIfc as SType
      extends SubjectIfc
    {...}
    public use WindowEvent as EventType;
  }

  public class WindowSubObsGrp extends SubObsGrp
    implements WindowSubObsGrpIfc
  {
    public static class WindowObserver as OType
      extends Observer implements WindowObserverIfc
    {...}
    public static class WindowSubject as 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.