Safe static type checking with systems of mutually recursive classes and inheritance. *

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

October 12, 1997

Abstract

Virtual classes, which were first used in Beta [MMP89], have recently been suggested by Thorup [Tho97] as an extension to Java to support genericity. While virtual classes are a convenient mechanism to support simultaneous modification of mutually recursive classes, they rely on dynamic type checking for type safety. We suggest a variant which has similar expressiveness as virtual classes, but supports safe static type checking. To ease the comparison with earlier work we express our construct as an addition to Java, though similar constructs may be added to a variety of object-oriented languages. For example, this proposal can also be seen as a statically type-safe alternative to Eiffel's "anchored types".

1 Introduction

Thorup [Tho97] recently suggested adding virtual types to Java in order to support genericity with a convenient and natural syntax. This construct, which is based on the virtual classes of Beta [MMP89], provides expressiveness similar to that of type parameters. We illustrate this construct with an example adapted from Thorup's paper. It is an implementation of the Observer pattern from [GRJV94].

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

class Subject{
    typedef OType as Observer;
    typedef EventType as Object;
    OType observers[];
    ...
    void notifyObservers(EventType e){
```

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

```
            int len = observers.length;
            for (int i = 0; i < len; i++)
                observers[i].notify(this,e);
        }
    }

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

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

The keyword `typedef` is used to introduce a type definition as part of a class definition. The advantage of this use of type definitions is that they may be overridden in a subclass by replacing the original type by a subtype (or subclass). It is well-known that the covariant redefinition of types can result in the loss of type safety in programs, but the flexibility allowed is often exactly what is desired.

For example, in the program above, `Observer` and `Subject` are mutually referential, while both depend on `EventType` (which is initially `Object`). Both are then specialized in order to deal with windows. The class `WindowObserver` needs to refer to `WindowSubject` rather than the less specialized `Subject` class. Similarly `WindowSubject` needs to refer to `WindowObserver` rather than `Observer` in order for new or overridden methods to make sense. Both need to refer to `WindowEvent`s rather than just plain `Event`s.

While the simultaneous covariant redefinition of `SType`, `OType`, and `EventType` is very useful for this example, it is well-known that this kind of covariant redefinition can lead to type errors. As a result Thorup proposes implementing virtual types by inserting dynamic type casts where expressions whose types depend on `typedef`'s are used. (Beta also uses dynamic type checking with its virtual classes.) Thorup argues that some of these dynamic casts can be eliminated in certain circumstances, though it is not clear from the paper what percentage of such casts can be eliminated.

Other proposals to extend Java to support genericity rely on the use of type parameters (see [OW97], [MBL97], [AFM97]), and [Bru97]. One advantage of virtual types over the proposals using type variables is that it is often awkward to model these covariant changes using only type parameters. As a result it appears that virtual types may be more useful than type parameters in examples like the one above that involve mutually referential classes which need to change consistently in extensions.

However the fact that the virtual types proposal requires dynamic type-checking has been seen as a significant disadvantage by many. A private mailing list including principals from the group defining Java and those proposing extensions of Java to support generics has supported discussions of the merits of the various proposals. As part of these discussions, Wadler observed the translation could be given several interpretations. To draw links with the existing proposals, he gave one translation into virtual types (as in Thorup's proposal), and a second into parametric types (as in Pizza and other proposals). The translation into parametric types provided for static type checking, but, as noted by Wadler, is more complex than one would like. Wadler also suggested that our earlier proposal [Bru97] for adding match-bounded polymorphism and `ThisType` to Java might be extended to support a similar construct, and we follow a program along those lines here.

In the rest of this paper we present a proposal for extending Java with a statically typed 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 $C_1, \ldots, C_n$ is a system of classes[1] 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 subclasses, $D_1, \ldots, D_n$ of the C_i such that all references to the C_i's are automatically changed to D_i's inside each of the subclasses. An important feature of this construct is that it can be statically type-checked.

2 The simple case: Understanding `ThisType`

In our earlier work [BSvG95, BFP97, Bru97] we solved a similar problem with *self*-referential classes using a `MyType` or `ThisType` construct. In a class definition, `ThisType` stands for the type of the object being constructed. 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:

```
class List
{
  char _head;
  @ThisType _tail;
  List (char head, @ThisType tail) {_head = head, _tail =tail; }
  char head () { return _head; }
  @ThisType tail () { return _tail; }
  void setTail (@ThisType tail) { _tail = tail; }
}
```

`ThisType` is another name for the interface of the class being defined. Thus the declaration of `_tail` specifies that it has the same interface as objects generated by the class. The advantage of using `ThisType` in type declarations rather than `List` is that the meaning of `ThisType` changes automatically in subclasses. We can provide type checking rules which ensure that no type errors occur, even with these automatic changes. In particular the bodies of methods are checked with the assumption that `this` has type `@ThisType` and `ThisType extends List`.

The prefix `@` attached to object types indicates that a value must be exactly that type. 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'` has type `@A`, then the assignment $a' = b$ will be illegal since only values of exactly type `A` can be held in `a'`. See [Bru97] or [BFP97] for explanations of the need for exact types in handling *binary methods*, *i.e.*, methods with a parameter of type `ThisType`.

3 Generalizing `ThisType` to mutually recursive classes

We wish to provide a statically checkable language construct to define mutually referential classes, where the classes are expected to change in parallel in extensions. Since they refer to each other,

[1]Our proposal can also accomodate interfaces, but we omit these here for simplicity.

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.

Java 1.1 now includes "inner" or nested classes which will provide the framework for grouping class definitions together.[2] The only syntactic extension we need is an `as` clause to associate flexible names for each class. We associate a `ThisType`-style name to a class as follows:

```
class C as TT{ ... }
```

Here `C` is the name of the class, while `TT` is the name of the `ThisType` interface associated with the class.

We can rewrite class `List` above using the `as` construct as follows:

```
class List as TType
   {
      char _head;
      @TType _tail;
      List (char head, @TType tail) {_head = head, _tail =tail; }
      char head () { return _head; }
      @TType tail () { return _tail; }
      void setTail (@TType tail) { _tail = tail; }
   }
```

The only change made was to replace the implicit `ThisType` with the explicit `TType` using the `as` clause.

Mutually referential classes which need to refer to each other's `ThisType` will be grouped together as inner classes of a class. The following is a reworking of Thorup's example in the new notation:

```
class SubObsGrp{

    class Observer as ThisObserver{
        void notify (ThisSubject subj, ThisEvent e){...};
    }

    class Subject as ThisSubject{
        @ThisObserver observers[];
        ...
        notifyObservers(ThisEvent e){
            int len = observers.length;
            for (int i = 0; i < len; i++)
                observers[i].notify(this,e);
        }
    }

    use EventType as ThisEvent;
```

[2]Thanks to Kresten Thorup for suggesting the use of inner classes as an existing mechanism to support the grouping of related classes.

```
}

class WindowSubObsGrp extends SubObsGrp{

    class WindowObserver extends Observer{...}

    class WindowSubject extends Subject{...}

    use WindowEventType as ThisEvent;

}
```

The class `SubObsGrp` has inner classes `Observer` and `Subject`. The "`use`" clause provides a definition for `ThisEvent`, which appears in both `Observer` and `Subject`. The `use` clause is provided to import a previously-defined interface which may be changed in a subclass of the "outer" class.

Type checking is done as follows. Inside `Observer`, `this` has type `@ThisObserver`, while inside `Subject`, `this` has type `@ThisSubject`. Because each class and interface may be replaced by extensions in a subclass of `SubObsGrp`, we only assume `ThisObserver extends Observer`, `ThisSubject extends Subject`, and `ThisEvent extends EventType` when type-checking the classes.

The subclass `WindowSubObsGrp` contains extensions of both `Observer` and `Subject`. Note that we do not repeat the declaration of the `as`-clause names from the superclasses, we just continue to use the same ones as in the superclass. The `use` clause redefines `ThisEvent` as the externally defined `WindowsEventType`. This time both classes are type checked assuming that `ThisObserver extends WindowObserver`, `ThisSubject extends WindowSubject`, and that `ThisEvent extends WindowEventType`. If an inner class is not extended in an extension of the outer class, the default is that that the old one is inherited unchanged.

By collecting the classes as inner classes of another class, we are informing the compiler that they all need to be type checked together. I.e., the assumptions on all types named in `as` clauses are needed in all of the component classes.

We access the inner classes by qualifying their names with the enclosing class. Looking forward to their uses later with match-bounded polymorphism, we'll refer to the inner classes using their `as` names qualified by the external class name:

```
class UseSubjectObserver {
  void doSomething (SubObsGrp.ThisEvent e) {
    @SubObsGrp.ThisObserver sub = new SubObsGrp.ThisObserver(e);
    ...
    sub.notifyObservers(e);
  }
}
```

4 Bounded polymorphism with mutually recursive groups

We can use these constructs with bounded polymorphism as usual, 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 `SubObj` to use `Event` as a type parameter.

```
class ParamSubObsGrp<Event extends Object>{
```

```
class Observer as ThisObserver{
    void notify (ThisSubject subj, Event e){...};
}

class Subject as ThisSubject{
    ThisObserver observers[];
    void addObserver(ThisObserver ob){...}
    notifyObservers(Event e){
        int len = observers.length;
        for (int i = 0; i < len; i++)
            observers[i].notify(this,e);
    }
}
```

Alternatively we could have had each of the inner classes independently parameterized by types.

We can also use the "outer" class as a type constraint, as in

```
class doSomething<T extends SubObsGrp>{
   ... T.ThisSubject ... T.ThisObserver ...}
```

Because of this use in polymorphic classes and methods, it will likely be more useful to use the names used in the as clauses, rather than those actually given inside the classes.[3]

5 Conclusion

The construct proposed here should allow one to type check most (perhaps all?) of the type-safe constructions available using virtual types. The major contribution is not the new syntax as such – it clearly borrows from existing inner classes and the ideas from Thorup's, Wadler's, and others' proposals – instead it is the idea that we can safely and statically type check these constructs with typing rules that are clearly understandable by programmers. There are many variations possible on the details of syntax for these constructs which should possess essentially the same type-safe static type-checking rules.

Similarly, other object-oriented languages could benefit from constructs similar to those proposed here. In particular, this proposal can also be seen as a statically type-safe alternative to Eiffel's [Mey92] "anchored types", as it allows the same covariant redefinition of types as in Eiffel; providing essentially the same expressiveness as these anchored types (though at the cost of identifying which types are available to be used as anchors).

The proof of the type-safety of this construct is a generalization of that given for our earlier language $\mathcal{LOOM}$ [BFP97].

References

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

[3]In fact, it might make sense to drop the distinct names altogether and keep only the "as" names.

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

[Bru97] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. 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 *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[GRJV94] E. Gamma, R.Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs*. Addison-Wesley, 1994.

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

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

[MMP89] O. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89 Proceedings*, pages 397–406. ACM SIGPLAN Notices,24(10), October 1989.

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

[PS94] J. Palsberg and M. Schwartzback. *Object-oriented type systems*. Wiley, 1994.

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

A An example with alternating lists

The following can be ignored by the program committee. We include it only to provide a more detailed illustration of the use of our new construct.

This example, based on an example from [PS94], supports lists whose nodes alternate between being from `XList` and `YList`. It can be seen as a generalization of the class `List` from the body of the paper.

```
class AltListGroup{
    class XList as XThis
    {
      char h;   // head
      @YThis t; // tail -- nb. use of YThis instead of YList
      XList (char _h, @YThis _t) { super(); h = _h, t = _t; }
      char head () { return h; }
      @YThis tail () { return t; }
      void setTail (@YThis _t) { t = _t; }
```

```
    }

    class YList as YThis
    {
      float h;  // head
      @XThis t; // tail -- nb. use of XThis instead of XList
      YList (float _h, @XThis _t) { super(); h = _h, t = _t; }
      float head () { return h; }
      @XThis tail () { return t; }
      void setTail (@XThis _t) { t = _t; }
    }
}

class LenAltListGroup extends AltListGroup
{
    class LenXList extends XList
    {
      int l;    // length
      LenXList (char _h, @YThis _t) {
        super(_h, _t);
        if (t == null) { l = 1; } else { l = 1 + t.length(); }
      }
      int length () { return l; }
    }

    class LenYList extends YList
    {
      int l;    // length
      LenYList (float _h, @XThis _t) {
        super(_h, _t);
        if (t == null) { l = 1; } else { l = 1 + t.length(); }
      }
      int length () { return l; }
    }
}
```

The extension `LenAltListGroup` adds the capability of providing the length of each list. The following is an example of the use of these classes.

```
class Breakit {
   void breakit () {
     AltListGroup.XThis xl = new LenAltListGroup.XThis('a', null);
     @AltListGroup.YThis yl = new AltListGroup.YThis(3.142, null);
     xl.setTail(yl);  // Wrong!
   }
}
```

The line marked *wrong* fails because we send a “binary method”, *setTail*, to a non-exact type. That is, because the type of the parameter of *setTail* is a `ThisType` construct, the receiver must be an exact type (just as in my original Java proposal). It can be fixed by changing the type of `x1` to `@AltListGroup.XThis`.