Family Genericity

Erik Ernst

Department of Computer Science, University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract. Type abstraction in object-oriented languages embody two techniques, each with its own strenghts and weaknesses. The first technique is extension, yielding abstraction mechanisms with good support for gradual specification. The prime example is inheritance. The second technique is functional abstraction, yielding more precise knowledge about the outcome. The prime example is type parameterized classes. This paper argues that these techniques should be clearly separated to work optimally, and also that current languages fail to do this. We have applied this design philosophy to a language based on an extension mechanism, namely virtual classes. As a result, some elements based on functional abstraction have been introduced, but they are simple and only used for things where they excel; conversely, the virtual classes have become more flexible, because their role is now more well-defined. We designate the result as *family genericity*. The presented language design has been implemented.

1 Introduction

People are different—for some, the world is full of possibilities to explore; for others, it is like a dangerous mountain precipice where every step must be secured. In general it is probably best for all people to be able to explore freely when that is safe, and watch every step when that need arises; but it is less likely to be a good idea to habitually mix the two modes. In the realm of programming language design some combinations of abstraction mechanisms actually force developers to constantly mix these two modes.

This paper focuses on two broad classes of object-oriented type abstraction mechanisms, one based on extension and the other based on functional abstraction. Extension lies at the heart of inheritance and subtyping. Traditional inheritance is hardly powerful enough to express statically typed contemporary software, but enhanced with virtual classes it is a viable platform.

Functional abstraction over types has also been known since the sixties in the shape of parametric polymorphism [26]. In System F, Girard [13] explicitly applied functions to types, but even in languages such as Standard ML [20] where these applications are implicit, the properties of the type system are highly influenced by it. Recently such mechanisms have become main-stream in statically typed object-orientation, due to the inclusion of type parameterized classes in Java 1.5 [14] and C$^{\#}$ 2.0 [19].

The fundamental structure of an extension mechanism is that a value from a given domain A is extended by a value from a different domain B, producing a new value from the original domain A, i.e., $A \rightarrow B \rightarrow A$. The prototypical example is that inheritance produces a new class (e.g., `ColorPoint`) by extending a given class (`Point`)

with an incremental entity (a class body { `String color; ...` }). In this case A is the domain of classes and B the domain of incremental entities (with a slight generalization: mixins [2,11]). The crucial point is that the outcome belongs to the same domain as the entity being extended.

The fundamental structure of functions used for type abstraction is that a value from a given domain A is applied to a value from a different domain B to produce a value in B, i.e., $A \rightarrow B \rightarrow B$. The prototypical example is that a parameterized type may be applied to a type, yielding a type. In practical language design there is often a shortcut such that parameterized classes are applied to type arguments to produce a class, but an implicit coercion from classes to types takes place here such that the fundamental structure is still $A \rightarrow B \rightarrow B$ at the level of types. Note that this is in fact required in order to allow type applications to be used as type arguments (as in `List<List<String>>`). The crucial point is that the outcome belongs to a different domain than the entity being applied.

The insight which has guided this work is that extension mechanisms by staying within one domain have great potential for flexible and repeated application. In return for this freedom it is both hard and inappropriate to try to establish guarantees about extensions that will *not* occur. Conversely, type functions are used to take one explicit step from the parameterized domain to the domain of non-parameterized end results, and the type system keeps track of whether or not this step has been taken. In return for this strict discipline, it is known for any given type application that information about the type arguments may be propagated into knowledge about the end result.

In summary, extension is *inclusive* because extensions can always be made, but the end result is only known to contain *at least* certain extensions. Type application is *exclusive* because it must occur exactly once, but the end result is then known to have *exactly* the given parameters.

The previous paragraph describes the ideal situation. In reality, the mechanisms have been polluted by attempts to give each of them the qualities which come naturally for the other, and it is a main point of this paper that they should be allowed to coexist in a clean form rather than overstretching each of them to do it all. The contributions of this paper are as follows:

- Identifying some fundamental structural differences between extension such as inheritance and virtual classes, and type functions such as parameterized classes.
- Concluding that extension be focused on unrestricted flexibility, and type functions should be focused on providing knowledge about the outcome.
- Revising the semantics and type system of the full-fledged programming language gbeta to follow these guidelines.
- In this process, introducing two new concepts, namely virtual class constraints and type-neutral mixins.
- Further developing the the so-called expression problem [16,25,9,28,30] to deal with the return type of a visitor, in a larger example that illustrates the redesigned language. As an aside, sketching a proposal for higher-order parameterized classes and type arguments.
- Implementing the resulting revised language design, and rewriting several hundred programs to use the revised language, thus providing some experience with the new language and honing its design along the way.

The rest of this paper is structured as follows: The next section briefly presents the two type abstraction mechanisms used as the starting point in this paper, and the following section outlines some well-known problems with them. Section 4 describes how a very simple notion of parameterized types (not classes) is added to a language design based on virtual classes. As a consequence, the virtual classes can then be simplified, as described in Sect. 5. The following section describes why this removal of features does not reduce the expressive power of the language. Section 7 proceeds to describe how the introduction of type-neutral mixins helps to separate types and classes in such a way that it is easier to write type-safe programs, and a larger example based on the expression problem is given to illustrate the properties of the resulting language. Finally, the implementation status is described in Sect. 8, related work is described in Sect. 9, and Sect. 10 concludes.

2 Existing Type Abstraction Mechanisms

Type parameterized classes in Java 1.5 will be used in this paper as the standard example of type functions. This mechanism makes it possible to equip classes with type parameters, to constrain them using bounds, to express co-/contra-variance in type applications by means of wildcards [14,29], and to enhance the expressive power of bounds by means of recursion, using so-called F-bounds [5].

```
class Box<X> {
  X x;
  X getX() { return x; }
  void setX(X x) { this.x=x; }
}
class ChineseBox<X extends ChineseBox<X>> extends Box<X>
{
  selfwrap() { setX((X)this); }
}
class ColoredChineseBox<X extends ColoredChineseBox<X>>
      extends ChineseBox<X> {
  String color;
}
class ChineseBoxF extends ChineseBox<ChineseBoxF> {
}
<X extends ChineseBox<X>> X unwrap(X cb) {
  return cb.getX();
}
```

Fig. 1. Boxes, expressed using Java parameterized classes

Figure 1 shows a parameterized class `Box` which uses its type parameter `X` in several ways. `ChineseBox` is a subclass of `Box`, intended to be able to contain a

`ChineseBox`. It propagates the type parameter to its superclass and also constrains it with an F-bound. The effect of this is that all type applications `ChineseBox<`T`>` are checked to ensure that the actual type argument T is a subtype of `ChineseBox<`T`>`. This is useful because it ensures that all applications of this parameterized class will produce a class with a certain recursive structure, namely that `X` denotes a type which is "similar" to `ChineseBox`. The class `ColoredChineseBox` illustrates that this technique can be applied at multiple levels in an inheritance hierarchy.

The polymorphic method `unwrap` illustrates that the F-bound succeeds in expressing that `getX` returns an object of a similar type as its argument, so for instance it is possible to access the `color` of the return value from `unwrap` applied to a `ColoredChineseBox`. However, the method `selfwrap` in `ChineseBox`, which makes the receiver wrap itself, must use a dynamic cast '`(X)this`' because the F-bound does not ensure that `this` is of type `X`.[1]

The class `ChineseBoxF` is a subclass of `ChineseBox` where the type argument is `ChineseBoxF` itself. This idiom is known as "taking the fixed point" of the parameterized class because it appears as the type argument to itself. Altogether, this is a standard technique (presented, e.g., in [3]) for using F-bounds to establish mutually recursive structures which can be maintained under inheritance.

This paper uses virtual classes in gbeta [6,7,8,10] as the target of the language redesign process. At this point virtual classes are briefly introduced, with examples that work in the language before the redesign.[2] Later examples will gradually show how the language was redesigned.

Figure 2 mimics Fig. 1 using virtual classes for type abstraction. Virtual classes were introduced in the late seventies with the language BETA [17]. They have been generalized in the language gbeta, and adapted to a Java-like context and in some ways clarified in the language Caesar [18]. A virtual class is a feature, similar to an inner class because it is a class and it is nested in an instance of the syntactically enclosing class, and similar to a (virtual) method because it is accessed using late binding. A virtual class is introduced by the keyword `virtual`. Subclasses of the enclosing class may redefine an inherited virtual class, and it must then be marked with the keyword `extended`. The effect is that the value of that virtual class becomes a subclass of its inherited value, computed by adding the contributions in the `extended` definition. This often amounts to the application of a mixin to the inherited value of the virtual class; in general it is a class combination operation defined by linearization. Redefinition of a virtual class replaces it with a subclass of its inherited value, so we use the word 'extension' rather than 'redefinition'. If the extension definition is marked with `final` then further extension of that virtual class in subclasses of the enclosing class is prohibited. This paper is intended to be self-contained, but because of the limited space we must refer to additional literature [6,7,8,10] for the detailed semantics and typing of virtual classes.

---

[1] In current Java implementations, this cast is 'unchecked' because the representation of types does not support a run-time check, but this is a problem with the type erasure implementation strategy, not with parameterized classes as such. E.g., C# does not use type erasure.

[2] For readability, the syntax of gbeta examples has been changed to follow the style of Java.

```
class Box {
  virtual class X extends Object;
  X x;
  X getX() { return x; }
  void setX(X x) { this.x=x; }
}
class CB {
  virtual class ChineseBox extends Box {
    final extended class X extends ChineseBox;
    selfwrap() { setX(this); }
  }
}
class ColoredCB extends CB {
  extended class ChineseBox { String color; }
}
aCB.ChineseBox unwrap(final CB aCB, aCB.ChineseBox cb) {
  return cb.getX();
}
```

Fig. 2. Boxes, expressed using gbeta virtual classes (transformed to a Java-style syntax)

Virtual classes provide type abstraction because a virtual class name (such as `X` inside `Box` in Fig. 2) denotes some subclass of its statically known value (here `Object`), but the actual value is determined by the extensions present at run-time. In order to establish the recursive structure of `ChineseBox` we make `ChineseBox` itself a virtual class and refer to it in the extension of `X`. As a result, `ChineseBox` is now a subclass of `Box` whose `X` is `ChineseBox` itself. The `selfwrap` method corresponds to the one in Fig. 1, but it does not need a dynamic cast. The receiver is known to have type `ChineseBox` or a subtype (`ChineseBox` is not a '`MyType`' [4], but it is known to be a supertype thereof). Next, `ChineseBox` is known to be equal to `X`, because the extension of `X` is final. Finally, `X` is again the argument type of `setX`, and hence `setX(this)` is type safe.

The enclosing class `CB` is needed in order to make `ChineseBox` a virtual class such that the relation between `ChineseBox` and its virtual `X` is preserved for extended versions of `ChineseBox`, e.g., the one in `ColoredCB`. Note that the recursive structure need not be redeclared when the color is added, as opposed to the situation with parameterized classes. Instances of `CB` and subclasses can be used to specify the type of `ChineseBox`, with whatever extensions the given subclass of `CB` has added. This phenomenon is known as *family polymorphism* [7,24,21]. For instance, the method `unwrap` accepts an argument `aCB` of type `CB`, and an argument `cb` of type `aCB.ChineseBox`, and the return value is of the same type. In particular, it is again possible to access the `color` of the return value from `unwrap` applied to a `ChineseBox` from an object known to be a `ColoredCB`.

Note that the type of the second argument `cb` depends on the *object* `aCB` provided as the first argument. Such a type would not be well-defined if it were possible to change

aCB during the life-time of cb, and the return type of the method also depends on aCB remaining unchanged throughout the method invocation. Because of this aCB is marked as final, which means that it cannot be changed after initialization. Whenever an object is used in this way to declare types of variables or method arguments, it is known as a *family object*, and the immutability restriction applies to all family objects. If a mutable variable were used to access a family object then the types declared from it would be useless, because no values can be shown to have those types.

3 Problems With Existing Mechanisms

The examples in the previous section already exhibit some problems which are characteristic of the two mechanisms. First, the method selfwrap in Fig. 1 contains a dynamic cast because the F-bound does not ensure any particular relation between X and the type of the receiver. Next, the method unwrap in Fig. 1 is essentially forced to re-declare the intended recursive structure between ChineseBox and its type argument X. It is an example of bad encapsulation that this supposedly internal recursive structure of ChineseBox must be restated whenever there is a need to use it or pass it on to another piece of code that uses it.

In comparison, the method selfwrap in Fig. 2 is safe and does not need the dynamic cast. The method unwrap receives the extra argument aCB which—like a "dynamic package"—provides the other types used in the method signature. This extra argument often causes irritation, but the notion of a dynamic package is quite simple after getting used to it, and it remains equally simple no matter how complex the contained structure is. In contrast, restating the recursive structure grows in complexity with the square of the number of members of the class family (in this case the family has just one member, ChineseBox), and may have specific quirks for some families. Moreover, virtual classes are more flexible because it is possible to work with family-polymorphic references to ChineseBox objects:

```
... final CB myCB = ...
myCB.ChineseBox cb1,cb2;
... cb1= new myCB.ChineseBox(); cb2.setX(cb1); ...
```

Ex. 1

In example 1 a family object myCB is in scope—it could be a final argument to a method, or a final instance variable of an enclosing object. By ordinary reference polymorphism it could be an instance of CB, ColoredCB, or any other subclass of CB. Using the type myCB.ChineseBox it is possible to declare variables of that specific ChineseBox type and operate safely on them because it is known to always be the same version of ChineseBox, namely the one in myCB. Hence, myCB works like a package because it provides access to a set of classes. There is no corresponding "package polymorphism" when using parameterized classes, because the class families consist of individual classes with no unified identification. It is, however, possible to create a polymorphic reference in Java using wildcards such that it is capable of referring to objects with more than one type argument T to ChineseBox<T>, as in the following example:

```
ChineseBox<?> cb1,cb2;
...cb2.setX(cb1); /* NO: not typable */
...cb1= new ChineseBox<?>(); /* NO: cannot use ? here */
```
Ex. 2

But there is no way wildcards in the type of these variables can express that the type argument of `ChineseBox` may vary at runtime and also that the type argument of `cb1` and `cb2` is the same. Consequently, expressions like `cb2.setX(cb1)` in example 2 are not type safe. It is also impossible to create new instances of `ChineseBox` in "the right family" because the '`?`' denotes an arbitrary type argument rather than the type argument given to `ChineseBox` in the type of `cb2` or any such thing.

In summary, type parameterized classes with F-bounds can go a long way to describe recursive type structures, but some problems remain, especially because the recursive structure is imposed from the outside rather than built-in.

Turning to the world of virtual classes, there are also some well-known problems. Collection classes using virtual classes for type abstraction have distinct type for each subclass created just in order to specify its element type. The `Box` class can be considered as a very simple example of a collection class, and we might then want to create some boxes containing numbers:

```
class NumBox extends Box
  { final extended class X extends Number; }
NumBox nb = new NumBox(); ...nb.setX(new Integer(3));
class NumBox2 extends Box
  { final extended class X extends Number; }
NumBox2 nb2 = new NumBox2(); ...nb2.setX(new Double(.5));
```
Ex. 3

The declaration of `X` as `final` in example 3 is important because it is needed to safely insert elements into the "collection", i.e., to call `setX()`. In a large software project there may be many different occasions where there is a need to create such a collection of the same type of elements. Each time a class like `NumBox` is created the result is a distinct class, unrelated to all the other classes which are also boxes holding numbers. Consequently, it is not possible to write generic code that polymorphically accesses a `NumBox` or a `NumBox2` and includes the information that the element type `X` is `Number`.

This is a nontrivial problem because it is hard to maintain the support for describing recursive structures and at the same time use structural type equivalence. In [27] it was in fact proposed to support a kind of structural equivalence for virtual classes, and Fig. 8 in that paper shows a family of mutually recursive classes with this kind of structural equivalence. However, no details are given, and several years of experience with the static analysis of gbeta suggests that, at least, the entire environment of enclosing scopes all the way out to the global namespace must be included in the structure which represents a type in order for this to work correctly. Nevertheless, if it works then it is a very interesting idea.

Another hard problem with virtual classes is that their inherently extensional nature makes it hard to reconcile them with lower bounds. In particular, it is difficult to use virtual classes to describe contravariance, i.e., that supertypes of a type argument create subtypes in type applications. Essentially, contravariance is useful in order to achieve

polymorphism over a type parameter `X` of a data structure that accepts method arguments of type `X` (an "`X` sink" data structure). E.g., if `List` is a parameterized type in Java and objects of type `T` should be inserted into such a list, but the actual type argument of the list should be allowed to be vary within the bounds of type safety, the proper type would be `List<? super T>` which uses `super` to specify contravariance and is allowed to refer to instances of `List<S>` for all types `S` such that `T` is a subtype of `S`. A detailed example of non-trivial and useful contravariance which can easily be expressed using wildcards has been given in a solution to the so-called expression problem [28]. The feature described in the next section enables gbeta to express this solution quite directly, too.

4 Adding Lightweight Type Parameters

This paper solves the problems described in the previous section by adding a simple version of type parameterization to virtual classes. As a consequence of this, virtual classes are simplified and made more flexible, as described in the next section.

The new type abstraction mechanism is based on *constraints* on virtual classes. It is only applicable to types, i.e., the declared type of an instance variable, a local variable, or a method argument, and in particular they cannot be applied to class definitions. They are not higher-order, there is no support for aliasing, and they do not allow F-bounds. The rationale for these design choices is that virtual constraints should be able to express certain typing properties known from parameterized classes with wildcards as in Java, but they should not be used for the specification of recursive type structures, because virtual classes are better at that anyway. Syntactically, virtual constraints are similar to type applications in Java:

```
class List { virtual class X extends Object; ... }
List<X extends Number> ro_nums; // covariance
List<X super Number> wo_nums;   // contravariance
List<X equals Number> rw_nums;  // invariance
```

Ex. 4

To check these types for correctness, it is required that `List` is statically known to contain a virtual class named `X` and the right hand side of each constraint is again a correct type. The main difference to Java type applications is that the constrained virtual is mentioned by name, e.g. `X` in the example, whereas wildcards are denoted by question marks and identified with specific type arguments by their position in the argument list. This is a natural consequence of the fact that virtual constraints are not passing arguments, they specify relations that must be verified to hold for the actual virtual classes in question. This also makes it possible to give multiple constraints on the same virtual class, and it enables constraining a subset of the virtual classes. When there is exactly one virtual class, an equality constraint on that virtual class is the default; e.g., the last line in example 4 could have used `List<Number>`. The names of the declared variables have been chosen to support a useful intuition about variance, namely that a covariant data structure is read-only, a contravariant data structure is write-only, and only an invariant data structure allows both writing and reading.

These variables can be assigned to each other according to the rules for use-site variance [15] which can also be used to explain the treatment of wildcard types in Java (when disregarding wildcard capture [29]). The following examples cover the cases which must be added to the subtyping rules in order to relate different variances:

```
wo_nums = ro_nums: /*ERR*/    rw_nums = ro_nums; /*ERR*/
ro_nums = wo_nums; /*ERR*/    rw_nums = wo_nums; /*ERR*/
ro_nums = rw_nums; /*OK*/     wo_nums = rw_nums: /*OK*/
```

Ex. 5

We can also verify that constraints are satisfied based on direct knowledge about virtual classes, especially because creation of a new object is as monomorphic as the class denotation (e.g., the object returned by '`new C()`' is known to be an instance of `C` whereas a variable of type `C` is only known to refer to an object of type `C` or a subtype). In particular, it is possible to assign the variables as follows (assuming that `Integer` is a subtype of `Number` which is again a subtype of `Object`):

```
ro_nums = new List() {extended class X extends Integer};
wo_nums = new List() {extended class X extends Object};
rw_nums = new List() {extended class X extends Number};
```

Ex. 6

Note that there is no need for a `final` modifier on the virtual class here, because the newly created object is known to be an instance of the denoted (anonymous) class which means that there cannot be any additional extensions of `X` in that object.

The introduction of virtual constraints immediately solves the problem of accidentally incompatible collection types, because the constrained types are structurally equivalent even though the underlying classes may be distinct. Virtual constraints can also immediately be used to express contravariance as with `super` bounded wildcard types. The structural equivalence can of course be extended to multiple levels by using nested constraints, as in the following example:

```
List<List<Number>> numss;
List<X extends List<X super Number>> ro_wo_numss;
```

Ex. 7

This declares `numss` to be a list of list of `Number`, and `ro_wo_numss` to be a read-only list of write-only lists of numbers. With the latter it is possible to iterate over the outer list and insert numbers into each of the inner lists. In fact, an inner list might be a `LinkedList<Object>` because the outer list might actually be a list of `LinkedList<X super Number>`, which can hold a `LinkedList<Object>`.

The scope rules are such that the constrained virtual classes are looked up in the base class of the nearest enclosing type—e.g., `X` is looked up in `List` in the examples—and the right hand side of the constraints are looked up in the same scope as the entire type. This "global" lookup rule also contributes to the simplicity of this mechanism, and ensures that recursion is handled using virtual classes rather than virtual constraints.

The language design decision to add virtual constraints to a language based on virtual classes can now be evaluated and motivated. First, we claim that type parameterization of classes does not interact well with inheritance because the two mechanisms are fundamentally different: inheritance is based on extension and class parameterization is based on type functions. Hence, it did not seem attractive to add type parameters

to classes—which would also produce a very complex and redundant language design. However, a coercion from class to type takes place—implicitly, but at well-known syntactic locations—for the declared type of each instance variable, local variable, and method parameter, and this process fits very well with type functions because it is inherently a single step process. Moreover, the addition of virtual constraints does *not* add much complexity to the language; the gbeta static analysis already computed all the information needed to determine whether virtual constraints are satisfied for assignments from expressions whose type does not have virtual constraints, and the variance rules used to determine subtyping among types with virtual constraints are very simple. There is a certain added complexity in the grammar, but the new constructs should be quite easy to understand for most programmers, and they are (even in the actual gbeta syntax) modeled to be syntactically similar to type application in main-stream languages.

As a result, the required amount of structural type equivalence is now available for the types of variables and arguments, in a familiar syntax, with variance that corresponds to Java wildcards. Finally, the new features are tightly integrated with virtual classes because it is virtual classes which are constrained, and because the right hand sides of constraints can also refer to virtual classes.

5 Simplifying Virtual Classes

It is tempting to consider language design as a matter of inventing new and sophisticated language features. However, it has always been a core design criterion for the language BETA that there should be few language features; they should be powerful each of them; they should work well together; and they should be orthogonal, i.e., their application areas should not overlap. The language gbeta was created in the community and spirit that created BETA, and simplicity and orthogonality are still very fundamental ideals. Hence, we consider it worthwhile and interesting to *remove* features during a language design process, and this is exactly what is described in this section: final extensions of virtuals are removed from the language.

As mentioned in Sect. 2, a virtual class extension definition marked with `final` implies that no more extensions to this virtual class can be made in subclasses of the enclosing class. This implies that more information can be established about the value of this virtual class: Normally, a virtual class is only known by upper bound when the class of the enclosing object is not known exactly; for instance, in the class body of `Box` in Fig. 2 it is known that `X` denotes some subclass of `Object`, but it cannot be assumed that any given type is a subtype of `X` because it is only bounded from above. (The type of `null` may be considered a 'bottom' type, but that is the only exception). Consequently, a variable or method argument of type `X` cannot be assigned any value unless it is already of type `X`. For a collection, e.g., this means that we can reorganize contained objects, but we cannot insert objects obtained from elsewhere. Final definitions of virtual classes thus play an important role of making it safe to assign externally provided objects to variables or method arguments whose type is a virtual class. Lists, say, that we cannot put anything into (except `null`) are not so useful.

However, the addition of types with virtual constraints as described in Sect. 4 immediately solves this problem in almost all cases: The collections and other objects for

which it is important to have a lower bound on a virtual class should be accessed through variables whose type has a lower bound on the virtual, e.g., `List<X super Number>` or `List<X equals Number>`. Using an `equals` bound is the typical choice for important references to such an object, because it preserves the possibility to both deliver and receive objects whose type is the virtual class (e.g., giving method arguments, and receiving method return values).

The remaining—hard—case is exemplified by `ChineseBox` in Fig. 2. In this case it is not possible to use a type with virtual constraints to establish the required typing properties because the constraints are concerned with an object rather than virtual classes of an object, and moreover that object is the current object, `this`. If we were to declare a suitably typed variable and assign `this` to it we would just move the typing problem to that assignment.

The `final` extension of `X` ensures that it is exactly equal to `ChineseBox`. This is because the contributions to `X` are known to be `Object` and `ChineseBox`, and the combination of these two contributions is statically known to be `ChineseBox`, even though this is a virtual class and its actual value is not known statically (just like $\emptyset \cup A = A$ for any set A). Since the current object `this` in any class body on the right hand side of a virtual class definition is an instance of that virtual class or a subclass thereof, we conclude that `this` has type `ChineseBox` and hence also type `X`.

If we remove the `final` keyword from the extension of `X` in `ChineseBox`, it is no longer known to hold that `this` has type `X`, which means that the method call `setX(this)` would be rejected by the compiler because it would not be type safe. It is easy to create an example showing that it would be unsound to consider it type safe, so this is an essential difference rather than a matter of improving the type analysis. A dynamic cast would have to be used, and the situation would then be just as bad as it is with parameterized classes.

However, there is a quite general approach which makes it possible to achieve the effect of a final declaration of a virtual class, and this is the topic of the next section. Based on this opening, virtual classes were simplified as follows: It is no longer possible to use `final` in a virtual class extension; a language mechanism known as 'disownment' [6, p.197] which was used to avoid multiple conflicting final extensions of virtual classes is no longer needed and is removed from the language; finally, quite a number of complex issues in the static analysis of gbeta are now gone, and the implementation of the type checker has been simplified correspondingly.

6 Emulating Final Declarations

Final extensions of virtual classes can be emulated based on a surprisingly simple mechanism, namely that of binding a final instance variable to an object, i.e., binding a name to a simple, opaque run-time value which is an address in the heap.

First, note that binding a name to a simple value is inherently a one-step process. This means that extension is wasteful because there will not be "multiple extensions". The language gbeta has had such a one-step binding mechanism for several years [6, p.193], known as 'virtual objects'. This mechanism is in fact what is used to express `final` method arguments such as `aCB` in Fig. 2 in the original gbeta syntax, but the

semantics is simply that of binding a name immutably to a value. Syntactically, virtual objects are different from method arguments, because they are declared as features of a class. The class and method concepts are in fact unified to one concept in BETA and in gbeta, namely *patterns*. So virtual objects are really features of objects created as instances of patterns, and it is a matter of personal taste whether one wants to consider a given pattern as a class or as a method, and correspondingly its instances as objects or as method activation records. For simplicity, virtual objects will be shown as features of classes, and the relation to methods is not made explicit.

A virtual object is introduced in a declaration marked with `virtual`, and for each virtual object introduction there must be at most one virtual object final declaration. If there is no final declaration of a virtual object the declared class of the object given in the introduction is used to create a new instance (it turns out to be convenient to get fresh objects by default), and if there is a final declaration then it specifies an object, which must be of the type given in the declaration or a subtype thereof. Here is an example:

```
myCB.ChineseBox myChineseBox;
...
class StickyBox {
  virtual object bx isa Box;
  bx.X x;
}
class MyStickyBox extends StickyBox {
  final object bx is myChineseBox;
}
```

Ex. 8

In this example, instances of `StickyBox` contain a virtual object of type `Box` or a subtype thereof, as well as an ordinary mutable instance variable `x`. The subclass `MyStickyBox` contains a corresponding `final` declaration of the virtual object, which binds the name `bx` in instances of that class to the object `myChineseBox` (assuming `myCB` is declared as in example 1). It is essential for the typing that a virtual object is immutable, because this makes it possible to use it as a family object. For example, `x` is declared to have the type `bx.X`, which means that it is known to be safe to execute statements like `x=new bx.X()`, `bx.setX(x)`, and `x=bx.getX()`.

Moreover, `x` has type `myChineseBox.X` in `MyStickyBox` , which establishes that it is also safe to execute `x.setX(x)` ... except that this only holds when the extension of `X` in `ChineseBox` is `final`, and we are in the middle of reconstructing that property after having removed `final` extensions from the language.

Note that the virtual object mechanism could as well have been expressed as a more traditional type function mechanism: A class having a virtual object could be considered as a function from objects to classes, which could then be managed in the type analysis in the usual way, and final declarations of virtual objects could then be application of this function to a given object. A class having multiple virtual objects would be a curried function (which has its own problems, as we shall see). Maybe this design would also have nice properties, but we have chosen the above described design because it enables the default mechanism without complex (implicit) type coercions, it fits well with the

rest of the language, and it maintains unification (there are only patterns and objects—not functions and patterns and objects).

Original Code

```
class C {
  virtual class X extends Number;
  X x; ...
}
class D extends C {
  final extended class X extends Integer; ...
}
```

Transformed Code

```
class Xholder { virtual class X extends Number; }
class C {
  virtual object Xh isa Xholder;
  Xh.X x;
  ...
}
class D extends C {
  final object Xh is
    new Xholder() { extended class X extends Integer; }
  ...
}
```

Fig. 3. Emulation of a final extansion (two-level)

The emulation uses the transformation shown in Fig. 3. Assume that a class `C` is given that introduces a virtual class `X` and uses it (in the example: as the type of an instance variable `x`). The subclass `D` of `C` contains a `final` extension of `X`. There are no ordinary (non-`final`) extensions of `X`. The transformation expresses an equivalent situation without using `final` extensions of virtual classes. The transformed declarations use a virtual, and the virtual class will then be looked up in that virtual object, as shown in the transformed code.

The only difference is that the virtual class is now wrapped inside the virtual object. The virtual object is managed by an application based mechanism, which means that the transition from unbound to bound is registered by the type analysis and will occur exactly once (by at most one final declaration or else using the default). This makes it possible to conclude that any given `final` declaration of a virtual object is the only such declaration for that attribute, and hence whatever is known about that particular object will hold for the attribute. In particular, it is known for all instances of `D` or subclasses thereof that their `Xh` virtual object is an instance of the given anonymous class (and not a subtype thereof!), and this means that `Xh.X` is exactly `Integer`. In other words, `Xh.X` was introduced as being some (unknown) subtype of `Number` in `C` and then fixed to be exactly `Integer` in `D` and its subclasses, just like `X` in the

original code. Note that the default binding of the virtual object corresponds to using the declared bound as a default for a type parameter in type parameterized classes.

In essence, we use a virtual object to achieve strict two-phase abstraction, then use it as a family object to provide the virtual class. The technique can be used to emulate all cases where a virtual class is used for introduction and `final` extension, which corresponds closely to the two-phase abstraction of a type parameter—first declaration of the formal, later passing actual arguments. Note that one virtual object may provide several virtual classes, possibly a mutually recursive family.

The technique does not directly handle cases where a virtual class in the version of gbeta before the redesign was introduced, then extended a non-zero number of times, and then `final` extended. However, this can be emulated for any pre-chosen number of levels, here three levels:

Original Code

```
class C1 {
  virtual class X extends D1;
  X x;
}
class C2 extends C1 {
  extended class X extends D2;
}
class C3 extends C2 {
  final extended class X extends D3;
}
```

Transformed Code

```
class XholderD1 { virtual class X extends D1; }
class XholderD2 extends XholderD1 {
  extended class X extends D2;
}
class C1 {
  class useHolder2 {
    virtual class X extends Holder2.X;
  }
  virtual object Holder2 isa XholderD1;
  virtual object Holder isa useHolder2;
  Holder.X x; // Holder.X <= Holder2.X <= D1
}
class C2 extends C1 {
  final object Holder2 is XholderD2;
  // Holder.X <= Holder2.X = D2
}
class C3 extends C2 {
  final object Holder is new useHolder2() {
   extended class X extends D3;
  }
  // Holder.X = D3
}
```

Ex. 9

We use one fewer virtual objects than the desired number of levels, i.e., two virtual objects in this case. Each level is then achieved by fixing one virtual object. Obviously, this technique is not practical for a large number of steps, but the simple two level emulation in Fig. 3 seems to suffice in practice. Several hundred small programs (of about 20.000 lines of gbeta code in total) were redesigned to use the revised language, and the simple two-level technique was sufficient in every case.

Static checking of virtual objects in the gbeta compiler is currently based on considering every program location where two classes are combined (including the degenerate case where a single mixin is added to a class). This includes every virtual extension declaration and every explicit class combination operation (explicit class combination is a kind of multiple inheritance that gbeta supports). If the two classes being combined are statically known then the combination can be performed statically and the result checked directly. If one of the classes is statically known and does not contain any `final` virtual object declarations then the combination cannot create conflicts, and the operation is accepted. In the remaining cases a warning is issued, and the merging operation is checked dynamically. Note that every program in this paper compiles without warnings.

Finally, it is possible that it would be useful to have syntactic sugar to emulate final virtual extensions as described in this section. It would have made the remaining examples in this paper a little bit more concise, but we have chosen not to do so in order to show the actual language design.

7 Introducing Type-Neutral Mixins

Over the years, the language gbeta has achieved more and more strict static analysis. Recently, the core of the static analysis of gbeta and Caesar [18] has been proven sound [10], which illustrates that significant progress has been made. However, a source of run-time errors that gbeta has always included is that of failing dynamic class combination. Since multiple inheritance in a statically typed language is often restricted to be a compile-time operation, it is maybe not surprising that it may fail if done at run-time. Nevertheless, gbeta has always supported it as a run-time operation, and has been progressing closer and closer to static safety. Note that the resulting class has always been a statically type safe class, just like the operation `a/b` may cause a 'DivideByZero' error at run-time, but if it succeeds then the result is known to be of type `int`, say, whose usage is statically type checked. As described at the end of the previous section, combination of classes which are not known statically will cause a warning, and a run-time error may occur at such a location (because of conflicting final declarations for a virtual object, or because of multiple mixins with the same declaration syntax but different enclosing objects). Because of this, it is very valuable to express programs such that class combinations operate on statically known classes wherever possible.

A very significant step in this direction has been taken by the introduction of type-neutral mixins. Note that it is a tempting simplification to remove ordinary nested classes from the language and just have virtual classes—seemingly, a virtual class can do everything that an ordinary class can do, and then some. But this is not a good idea

because a large amount of static knowledge is lost—there are basically no lower bounds on types any more, so most assignments and method calls are no longer type safe.

However, it is no problem to replace an ordinary nested class by a virtual class if the type of that class is fixed and statically known. In other words, extensions can be made as long as they do not affect the typing properties of the class. This is exactly what characterizes type-neutral mixins: type-neutral mixins can be added to a class, but the resulting class has the same type as the original.

Consequently, the language *was* in fact simplified, but then immediately extended a little again: Ordinary nested classes were removed from the language, and type-neutral mixins were added. Moreover, each virtual class is open for general extension, or open for type-neutral extension only. We use the terms *type virtual* and *implementation virtual* to distinguish the two. Type virtual classes are marked with the keyword `virtual` (so they look like the old virtuals), and implementation virtual classes do not have this keyword (so they look like the old ordinary classes, and work the same at introduction, but now their implementation can be extended). Consequently, it is possible to use statically known types in all those locations where the desired extensions are only concerned with implementation, and this turns out to be a quite common situation. The following rules govern type-neutral mixins:

1. A mixin is type-neutral iff it occurs on the right hand side of an `extended` virtual class declaration and the corresponding introductory declaration is not `virtual`.
2. A type-neutral mixin cannot introduce a type virtual class or a virtual object.
3. A type-neutral mixin can only extend inherited virtual classes with type-neutral mixins, and cannot contain a `final` virtual object.
4. Features in a type-neutral mixin can only be accessed from itself and nested scopes.

A couple of comments should be added to these rules: Ad (1), it might be useful to be able to create type-neutral mixins elsewhere, e.g., in an anonymous class, but the need has not arisen so far. Ad (3), note that both kinds of virtual classes can be extended with type-neutral mixins. Ad (4), note that this implies that no features introduced in a type-neutral mixin can be accessed by subclasses or client code, i.e., everything inside a type-neutral mixin is private. On the other hand, code in a type-neutral mixin can freely use features inherited from other (non-type-neutral) mixins.

The most complex issue in relation to static analysis of type-neutral mixins is that they must be included in the analysis of the code inside them, but they must be invisible during analysis of other mixins in the same object, and analysis of client code (anything outside the syntactic scope of the type-neutral mixin itself). The current approach in the gbeta compiler is to alwys include them, but ignore them for lookup from any location outsite the type-neutral mixin itself. Since the result of lookup during compilation strictly controls the lookup at run-time (gbeta has static name binding), it is impossible for a run-time lookup to select a feature from a type-neutral mixin unless the same thing occurred during static analysis, i.e., unless it originated from inside that mixin. Finally, type-neutral mixins are ignored for subtype comparison.

Note that the need for type-neutral mixins is more acute in gbeta than it seems when considering the examples in this paper. Because of the Java style syntax, methods and classes look different in this paper; but in the original gbeta syntax methods and classes are always patterns, i.e., the syntax and the semantics does not distinguish methods and

classes. This means that (virtual) methods have always been expressed using virtual patterns (in both BETA and gbeta), so they have been very common. With the introduction of type-neutral mixins most methods can now be implementation virtual, so a significantly larger proportion of types are now statically known.

To illustrate the possibilites in the redesigned language, consider the example in Fig. 4. This example assumes that the virtual class `X` in `Box` and `ChineseBox` from Fig. 2 has been transformed as in Fig. 3 to use a virtual object rather than a `final` extension of a virtual class. It declares two successively derived class families from `CB`. First, `IntCB` adds a method `accept` to `ChineseBox` and a subclass `IntBox` of `ChineseBox` wrapping an integer variable. It also adds the type virtual `Visitor`, which is used in an instance of the *visitor* design pattern [12], as well as a concrete visitor subclass `toStringVis`. Next, `IntPairCB` adds another member to the family, `PairBox`, wrapping two instances of `ChineseBox`, and extending `Visitor` and `toStringVis` to handle `PairBoxes`. Note that `toStringVis` by being implementation virtual avoids combining two classes which are not statically known. Also note that it is possible to implement the *extended* interface declared by extending the `Visitor` class with a new method `forPairBox` in `IntPairCB`. In other words, `toStringVis` is only open for type-neutral extension *on its own*, but since it is a subclass of a type virtual it can be subject to type extensions, namely type extensions of the type virtual superclass. Finally, we can use the class system as follows:

```
final IntPairCB IPCB = new IntPairCB();
IPCB.IntBox ibox = new IPCB.IntBox();
IPCB.PairBox pbox = new IPCB.PairBox();
ibox.i=3;  pbox.b1=pbox.b2=ibox;
System.println(pbox.accept(new IPCB.toStringVis()));
```

Ex. 10

It is not trivial to reconstruct a similar class system using type parameterized classes. In fact, we do not know whether or how it can be done. But if we assume that type parameterized classes are extended with a couple of new features then it is possible to get somewhat close. The new features are higher-order parameterized classes and higher-order type arguments. In light of the description of parameterized classes as functions given earlier this appears to be a rather obvious extension, but we are not aware of any proposals or implementations that directly covers this idea.

Where a type parameterized class is a function from a tuple of types to a type (and class), the extended language would accept functions over types as type arguments as well. Moreover, classes could be parameterized with more than one tuple of type arguments, providing functions from types to types to a type (and class), etc. We have assumed and used this hypothetical extension to Java in Fig. 5.

The reason why such an extension is required is that we wish to specify a class family of which one member is a parameterized class. This is because the return type of `accept` should be a type parameter of the given visitor, but we do not want to fix this parameter when fixing the family. For instance, we want to be able to create a fixed point of `IntChineseBox` by providing the type arguments `X` and `V`, and then choose the result type `R` for each concrete visitor in a separate type application step. Otherwise, if we must provide `R` when the family is created then the resulting class family will only be able to have visitors returning one type of result.

```
class XHolder { virtual class X extends Object; }

class IntCB extends CB {
  extended class ChineseBox {
    VIS.Rh.X accept(final Visitor VIS);
  }
  class IntBox extends ChineseBox {
    int i;
    VIS.Rh.X accept(final Visitor VIS) {
      return VIS.forIntBox(this);
    }
  }
  virtual class Visitor {
    virtual object Rh isa XHolder;
    Rh.X forIntBox(IntBox ibox);
  }
  class toStringVis extends Visitor {
    final object Rh is new XHolder() {
      extended class T extends String
    }
    Rh.X forIntBox(IntBox ibox) {
      return Integer(ibox.i).toString();
    }
  }
}

class IntPairCB extends IntCB {
  class PairBox extends ChineseBox {
    ChineseBox b1,b2; // NB: 'b1=this;' is OK
    VIS.Rh.X accept(final Visitor VIS) {
      return VIS.forPairBox(this);
    }
  }
  extended class Visitor {
    Rh.X forPairBox(PairBox pbox);
  }
  extended class toStringVis {
    Rh.X forPairBox(PairBox pbox) {
      return "(" + pbox.b1.accept(this) +
             "," + pbox.b2.accept(this) + ")";
    }
  }
}
```

Fig. 4. Adding visitors to the running `Box` example

```
abstract class IntChineseBox<X extends IntChineseBox<X,V>,
                             V extends IntVisitor<X,V>>
    extends ChineseBox<X> {
  abstract <R> R accept(V<R> vis);
}

class IntBox<X extends IntChineseBox<X,V>,
             V extends IntVisitor<X,V>>
    extends IntChineseBox<X,V> {
  int i;
  <R> R accept(V<R> vis) {
    return vis.forIntBox(this);
  }
}

abstract class IntVisitor<X extends IntChineseBox<X,V>,
                          V extends IntVisitor<X,V>>
                         <R> {
  abstract R forIntBox(IntBox<X,V> ibox);
}

abstract
class IntPairChineseBox<X extends IntPairChineseBox<X,V>,
                        V extends IntPairVisitor<X,V>>
    extends IntChineseBox<X,V> {
}

class PairBox<X extends IntPairChineseBox<X,V>,
              V extends IntPairVisitor<X,V>>
    extends IntPairChineseBox<X,V> {
  X b1,b2; // NB: type too small: 'b1=this;' is unsafe
  <R> R accept(V<R> vis) {
    return vis.forPairBox(this);
  }
}

abstract
class IntPairVisitor<X extends IntPairChineseBox<X,V>,
                     V extends IntPairVisitor<X,V>>
                    <R>
    extends IntVisitor<X,V><R> {
  abstract R forPairBox(PairBox<X,V> pbox);
}
```

Fig. 5. Adding visitors using (hypothetical) higher-order type arguments

As a consequence, all visitor classes are higher-order type parameterized classes because they need to receive the two first type arguments when the family is created, and then later receive the type argument which determines the result type of `accept`. The further consequence of this is that the type argument `V` is everywhere of kind type to type rather than just type, and this is also the reason why it is possible to use the expression `V<R>` in the signature of `accept`.

Apart from this, Fig. 5 follows the pattern from Fig. 4 with the adjustments needed for class families based on parameterized classes. It does not define a concrete visitor, but this will be discussed below. Example 11 shows how these class families can be used in a similar way as example 10:

```
abstract class IntPairChineseBoxF
  extends IntPairChineseBox<IntPairChineseBoxF,
                            IntPairVisitorF> {}
class IntPairIntBoxF
  extends IntBox<IntPairChineseBoxF,IntPairVisitorF> {}
class IntPairPairBoxF
  extends PairBox<IntPairChineseBoxF,IntPairVisitorF> {}
class IntPairVisitorF<R>
  extends IntPairVisitor<IntPairChineseBoxF,
                         IntPairVisitorF><R> {}
class toStringVis extends IntPairVisitorF<String> {
  String forIntBox(IntPairIntBoxF ibox) {
    return Integer(ibox.i).toString();
  }
  String forPairBox(IntPairPairBoxF pbox) {
    return "(" + pbox.b1.accept(this) +
           "," + pbox.b2.accept(this) + ")";
  }
}
IntPairIntBoxF ibox = new IntPairIntBoxF();
IntPairPairBoxF pbox = new IntPairPairBoxF();
ibox.i=3;  pbox.b1=pbox.b2=ibox;
System.println(pbox.accept(toStringVis));
```

Ex. 11

As always, we need to take fixed points before we have actual classes at hand (note how taking fixed points gets more complex with larger families, whereas the corresponding operation with virtual classes of using a family object has the same complexity for all kinds of families). More importantly, note that the concrete visitor is based on these fixed point classes. We have not found a way to express the concrete visitor as part of the specializable class families rather than as a subclass of the fixed point visitor, which means that concrete visitor classes cannot be specified incrementally in each family, nor can they be reused for other (parameterized or fixed) families. The problem is that the type parameter list `<R>` of all visitor classes must appear last in order to enable `accept` to access it using the expression `V<R>` and in order to create the families by fixing on the arguments `X` and `V`. We could try to swap the order of the type argument lists on visitor classes and reconstruct the class system to create concrete visitors before fixing the family, but this would not work because of the need to use `V<R>` in the signature of

`accept`. Even then, if it were possible to do this it would preclude the external (post-fixed-point) expression of concrete visitor classes like `toStringVis` in example 11. Note that it is trivial to write such a post-fixed-point visitor using virtual classes.

All in all, the usage of higher-order functions to describe gradual (here: two-step) parameterization is less flexible than extension, because two function applications will by nature be ordered one way or the other, whereas the order of extensions can be chosen freely. Perhaps explicit support for higher-order type functions or pattern matching could be used to manipulate the ordering of the type parameter lists in Fig. 5, but we have not explored this possibility.

In summary, type parameterized classes seem to be unable to express the incremental specification of concrete visitor classes before fixing class families, even when the language is hypothetically extended with higher-order parameterized classes and type arguments. Additionally, there is still the unsolved problem (which gave rise to the unsafe cast in `setX` in Fig. 1) of not having a subtype relation between `this` and the type argument intended to describe the "this family member". The instance variables `b1` and `b2` in `PairBox` in Fig. 5 exemplify this.

The last point we wish to make is concerned with dynamic type discovery, i.e., type casting. It was already mentioned in connection with Fig. 2 that family objects provide a single entity polymorphically identifying the relevant family of classes. In addition to the consequences that we have already described, this enables a kind of "large scale cast" operation which does not have an equivalent operation in context of type parameterized classes, even if type arguments are fully reified at run-time.

We have to briefly introduce type casting in gbeta first. There is no type cast operation, but there is a type case control structure which is thread-safe and enables the execution of code under explicitly stated enhanced typing assumptions, iff those assumptions turn out to be satisfied at run-time. (This is the `(when...when)` statement in the original gbeta syntax, but here we will only show a specialized example, adjusted to fit the syntactic style of Java). Here is an example:

```
boolean compose(final IntCB aCB1, final IntCB aCB2,
                aCB1.ChineseBox cb1,
                aCB2.ChineseBox cb2) {
  typeif (aCB1==aCB2) {
    cb1.setX(cb2); cb2.setX(cb1);
    return true;
  }
  else { /* Not same family! */ return false; }
}
```

Ex. 12

In example 12, the method `compose` receives two family objects `aCB1` and `aCB2` as well as two instances of `ChineseBox`, one from each class family. The body of the method contains a type case statement which works as follows: If `aCB1` and `aCB2` are the same object then the statement block after `typeif` will be executed, under typing assumptions as in the body of the method, *extended* with the information that `aCB1` and `aCB2` is the same object. Consequently, operations including expressions whose type depends on `aCB1` or `aCB2` will now be analyzed differently, and for example the two given invocations of `setX` are known to be safe. If `aCB1` and `aCB2` are not the

same object then the `else`-part of the statement is executed, under the same typing assumptions as in the method body.

Even using fully reified type parameterized classes with higher-order extensions, it seems difficult to dynamically establish such a large scale typing property. It is much more likely that there would have to be a separate cast operation at each point where an operation depends on having members of the same family, as shown in the following example:

```
<X1 extends IntChineseBox<X1,V1>,
 V1 extends IntVisitor<X1,V1>,
 X2 extends IntChineseBox<X2,V2>,
 V2 extends IntVisitor<X2,V2>>
boolean compose(X1 cb1, X2 cb2) {
  if (cb2 instanceof X1) {
    if (cb1 instanceof X2) {
      cb1.setX((X1)cb2);
      cb2.setX((X2)cb1);
      return true;
    }
    else { /* Not same family! */ return false; }
  }
  else { /* Not same family! */ return false; }
}
```

Ex. 13

Finally, note that there are two different ways in which it can be established that the two class families are distinct; this reflects the fact that programmers are not forced to use the type arguments `X` and `V` to `IntChineseBox` and `IntVisitor` in the style described as taking the fixed point. It is not obvious whether this extra flexibility is sometimes useful, but when describing mutually recursive families of classes it will invariably cause extra complexity in cases like this one.

8 Implementation Status

Apart from the fact that the original gbeta syntax has been transformed to a style similar to Java and all directives concerned with the module system have been left out, the example programs in this paper are actual, running gbeta code. There is one exception—in the current implementation, example 12 does not work as stated and must be expressed in a more verbose form using two intermediate local variables, but this is expected to be a rather shallow bug. The implementation of virtual constraints is more than a year old and is rather stable. The implementation of type-neutral mixins is much newer, but seems relatively stable, too. About 20.000 lines of gbeta code in about 650 programs (most of them small, but up to 2500 lines of code) have been updated along the way to use the revised language, thereby helping to evaluate the new language design. The implementation is available at `http://www.daimi.au.dk/~eernst/ecoop-gbeta/`.[3]

[3] This is a provisional release, but an official release containing all the recent extensions will be made available at `http://www.daimi.au.dk/~eernst/gbeta/` within a few weeks.

9 Related Work

Several related research efforts have been mentioned already, so at this point we just add a few extra remarks.

In [3] the relation between type parameterized classes and virtual classes is described as if the significant difference in the treatment of families of mutually recursive classes is in the verbosity of parameterized classes (and type safety, but that discussion has changed since then). This paper demonstrates that there are also some deeper differences between the abilities of these two approaches to express complex typing structures.

We mentioned that virtual constraints are similar to wildcards [14,29], but deferred discussion of wildcard capture. Wildcard capture is a mechanism that enables invocation of a polymorphic method on an argument whose type includes a wildcard as a type argument, effectively giving a name to the type argument which is otherwise only known as '`?`'. In fact, this capability is just a special case of the general ability of accessing "type arguments" (virtual classes) in a language with virtual classes—in such a language the type argument is never nameless.

The language SCALA [22,23] features abstract type members which share many propreties with virtual types (compared with gbeta they have been more formally well-described for years, but the run-time semantics is less rich). In essence, abstract type members must *always* have a final 'extension', because the 'introduction' and 'extensions' are supertype constraints and only when the type member is finally bound to a concrete class (which must satisfy all the constraints) it can be used for such things as creating new objects. Hence, this mechanism is clearly divided into two phases.

The expression problem has been the focus of several efforts over the years. Krishnamurthi et al. [16] describe an approach based on visitors where there is no notion of a class family, it is simply an ordinary hierarchy of 4 classes where a fifth class is then added and a subclass of the visitor created to handle also the fifth class. The added class expects the extended visitor, but instances of the other classes continue to use the type of the original visitor, so the type analysis does not ensure that every object will be visited by a visitor that knows how to handle it, which of course gives rise to run-time type errors. A solution based on factory methods is described; this prevents the run-time type errors, but because of programmer discipline rather than static type checking. Philip Wadler gave an influential formulation of the expression problem and presented a tentative solution in Pizza on the `java-genericity` mailing list [25], but it turned out to be impossible to realize, exactly because of the lack of a subtyping relation for `this` mentioned earlier in this paper. In [28] four approaches are presented using Java 1.5, including some based on visitors. In [30] many approaches in SCALA are presented, including several based on visitors. None of these deal with the problem of handling the return type of the method corresponding to `accept`: the return type of this method is everywhere `void` or `unit`, and visitors which must return a result use the work-around of storing it in an instance variable in the visitor.

Finally, it seems likely that the sketchy idea about adding higher-order constructs to the type parameterization of Java is related to the notion of higher-order functors in Standard ML [1]. However, there are no equality constraints on types in parameterized

classes, and there is no dynamic subtype polymorphism associated with Standard ML modules, so certainly the relation may be distant.

10 Conclusion

This paper argues that virtual classes and type parameterized classes belong to fundamentally different categories of type abstraction mechanisms, and that they should be kept separate in order for each of them to work optimally. Starting from the language gbeta which is based on virtual classes, this philosophy is applied in a language redesign process; a simple type parameter mechanism is added, the virtual classes are simplified by removing final extensions, and type-neutral mixins are added in order to preserve static knowledge about types without restricting the implementation of classes. The power of the resulting language is illustrated in an extension of the expression problem which turns out to require an extension of type parameterized classes with higher-order constructs, but still does not quite enable parameterized classes to express the desired typing structure.

References

1. Sandip K. Biswas. Higher-order functors with transparent signatures. In *Proceedings POPL'95*, pages 154–163, New York, NY, USA, 1995. ACM Press.
2. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, 10, pages 303–311, October 1990.
3. K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–549, 1998.
4. K. B. Bruce, R. Van Gent, and A. Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. *Proceedings ECOOP'95*, LNCS 952:27–51, 1995.
5. Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989. Also technical report STL-89-5, from Software Technology Laboratory, Hewlett-Packard Laboratories.
6. Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
7. Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.
8. Erik Ernst. Higher-order hierarchies. In Luca Cardelli, editor, *Proceedings ECOOP'03*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.
9. Erik Ernst. The expression problem, scandinavian style. In *Proceedings MASPEGHI'04, in assoc. with ECOOP'04*, 2004.
10. Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings POPL'06*, pages ?–?, Charleston, SC, USA, 2006. ACM. To appear.
11. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

13. Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).
14. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification – Third Edition*. The Java Series. Addison-Wesley, third. edition, 2004.
15. Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parameteric types. In Boris Magnusson, editor, *Procedings ECOOP'02*, LNCS 2374, pages 441–469. Springer-Verlag, June 2002.
16. Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing ojbect-oriented and functional design to promote re-use. In Eric Jul, editor, *Proceedings ECOOP'98*, volume 1445 of *LNCS*, pages 91–113, Brussels, Belgium, July 1998. Springer-Verlag.
17. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
18. M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings AOSD'03*, pages 90–99, Boston, USA, 2003.
19. Microsoft Corporation, Seattle, USA. *C# – Version 2.0 Specification*, May 2004.
20. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.
21. Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings OOPSLA'04*, pages 99–115. ACM Press, 2004.
22. Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
23. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings OOPSLA'05*, pages ?–? ACM Press, October 2005.
24. Klaus Ostermann. Dynamically composable collaborations with delegation layers. *Lecture Notes in Computer Science*, 2374:89–110, 2002.
25. The expression problem, November 1998. Message to the java-genericity electronic mailing list.
26. Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Programming Languages, Copenhagen, Denmark, 1967.
27. Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings ECOOP'99*, pages 186–204, 1999.
28. Mads Torgersen. The expression problem revisited – four new solutions using generics. In Martin Odersky, editor, *Proceedings ECOOP'04*, LNCS 3086, pages 123–143, Oslo, Norway, 2004. Springer-Verlag. To appear.
29. Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. *Journal of Object Technology*, 3(11), 2004. `http://www.jot.fm/issues/issue_2004_12/article5`.
30. Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2004.