Genericity in Java with Virtual Types

Kresten Krab Thorup

DEVISE – Center for Experimental Computer Science
Department of Computer Science, University of Aarhus
Ny Munkegade Bldg. 540, DK-8000 Århus C, Denmark
Email: krab@daimi.aau.dk

Abstract This paper suggests virtual types for Java, a language mechanism which subsumes parameterized classes, while also integrating more naturally with Java's object model. The same basic mechanism is also known as virtual patterns in BETA and as generics in ADA95. We discuss various issues in the Java type system, issues with inheritance and genericity in general, and give a specific suggestion as to how virtual types should be integrated into Java. Finally we describe how to make an efficient implementation of virtual types based only upon the existing Java virtual machine.

1 Introduction

Java is a new programming language which is interesting for many reasons. First of all, it is not the result of a language-research project in the traditional academic sense. Java is the result of engineering work, assembling many useful features developed for other programming languages, most visibly language features of SIMULA [6], Objective C [24] and C++ [8]. In fact, it has been an expressed goal in the design of of Java to stick with programming language mechanisms that have already been proven through serious use.

Another interesting aspect of Java is that it reintroduces "safe programming," throwing away much of its heritage from C++: pointer arithmetic is gone and language-level protection mechanisms (such as private and protected) are enforced in the execution environment, unlike in C++ where such mechanisms can be worked around. Safeness, in this sense that programs should not be able to "crash," is another corner stone of the design of Java.

As an emerging standard programming language, Java is still open to enhancements. As such, there are several extensions and enhancements [7, 12, 22, 25] being proposed and developed, and this paper addresses one area where enhancement is needed namely the need for genericity. There is a wealth of ways genericity could be implemented in Java, e.g. by using parameterized types as it is known from C++ and EIFFEL [21], or by using functional polymorphism, as in ML.

The rest of this paper will progress as follows: Section 2 briefly introduces the notion of virtual types. Section 3 reviews the Java type system, and Section 4 discuss virtual types in context of issues with genericity and inheritance in general. Section 5 provides a detailed description of our design, and Section 6 outlines how to implement virtual types and describes some of the performance characteristics. Section 7 discusses related issues and work. Section 8 concludes.

2 Virtual Types

Our work is based on the ideas of *virtual patterns* in the BETA programming language [14, 18, 19]. A decade later, a similar mechanism is also found as *generics* in ADA95 [29], and as *creators* in [28]. As such, what we are about to present has already been in use for more than a decade in various programming languages.

However, in context of Java the mechanism does present several interesting challenges, particularly in the implementation because we have to generate Java programs that would be accepted by the type-inference integrity check performed by the execution environment.

2.1 An Informal Introduction

With our language extension, class and interface definitions can be augmented with virtual type declarations, each introducing a new type name as an alias for some existing type. The details are described later, but the general idea is that "`typedef` *Name* `as` *Type*" introduces an alias for type *Type* named *Name*, much like in C and C++. In the following example, class `Vector` declares a virtual type named `ElemType`.

```
class Vector {
  typedef ElemType as Object;
  void addElement (ElemType e) ...
  ElemType elementAt (int index) ...
  ...
}
```

For instances of this `Vector` class it would make no difference if the virtual type declaration was removed, and `Object` substituted for `ElemType`.

In context of a subclass however, a virtual type may be *extended* to be an alias for some subtype of the type it was an alias for in the superclass. The effect of extending a virtual type is that all inherited entities that are qualified by the virtual type in context of the superclass, will adopt the local alias of the virtual type when used in context of the more specific class. Now consider a subclass of `Vector` which extends the virtual type `ElemType`, to be qualified by `Point` rather than `Object`.

```
class PointVector extends Vector {
  typedef ElemType as Point;
}
```

For all the methods and fields inherited from `Vector` to `PointVector`, the alias type `ElemType` will be referring to `Point` rather than `Object`. So `PointVector` defines a vector of elements, which is statically typed to hold instances of class `Point` or subtypes thereof. Thus, where one today is limited to use code of the form:

```
Vector v = new Vector();
v.addElement (new Point(2,2));
...
Point p = (Point) v.elementAt(0);
```

Using the new class `PointVector` above, we could have written the following, without the explicit cast in the last line:

```
PointVector v = new PointVector();
v.addElement (new Point(2,2));
...
Point p = v.elementAt(0);
```

To facilitate recursive class types, a special virtual type called `This` is automatically available in all classes. This special virtual type always refers to the type of the enclosing class. Thus, in any given context, the dynamic type of the special variable `this` will always be the special type `This`. To illustrate this behaviour, consider a linked list element class:

```
class Link {
  This next;
  This prev;
  void insertBetween (This p, This n) {
    p.next = this; n.prev = this;
    prev = p; next = n;
  }
  ...
}
```

To use this `Link` class, it is subclassed and some interesting behaviour is added. In context of the subclass, the special type `This` is bound to `StringLink`.

```
final class StringLink extends Link {
  String value;
  ...
}
```

After which instances of `StringLink` can only be linked to other instances of `StringLink`, because calling `insertBetween` for an instance of `StringLink` automatically asserts that both arguments are also instances of that same class.

3 Java's Type System

Since we are suggesting changes to Java's type system, we will briefly review and critique of the existing type system. This is a rather dense description, so we have included references to material where further examples can be found.

3.1 A "Riddle" of Type Systems

The type system of any object-oriented programming language will always reflect some tradeoff between the following three desirable properties: *covariance typing*, *full static typing* and *subtype substitutability*. It is not possible to have all three, since supporting any two of these mechanisms in full, will mutually exclude the third. A more complete discussion of this conjecture can be found in [16, 17], but we will briefly illustrate it here with an example:

Consider an `insert` method in a `GeneralList` class and in a `PersonList` class. One would want to be able to have the argument of `insert` be qualified with `Person` in the latter, and `Object` in the former. If a type system allows this, it is said to allow covariance typed methods, or simply covariance. This is a desirable property of a type system, because in general, a more specific class would naturally require more specific types of arguments. We would also like subtype substitutability, i.e. assuming `PersonList` is a subclass of `GeneralList`, a reference to a `PersonList` object can be stored in a variable qualified by `GeneralList`. Seeking to allow both of the above as well as static typing introduces a problem because the `insert` method invoked for a variable qualified by `GeneralList` only requires an `Object`, while in reality, a `Person` is needed if it is referring to the more specific `PersonList`.

This riddle of type systems apparently was not described until 1990 [32], but understanding it's implications lets us understand different type systems much better.

3.2 Java's Tradeoff

While Java is indeed a typed programming language, we do not consider it fully statically typed since the execution of Java programs may still uncover type errors in the following two situations:

- When making explicit "down casts" also known in the literature as reverse assignments [11, §15.15]. This is allowed by the compiler if it cannot be statically determined that the cast will always go wrong. Such casts must be checked at runtime.
- When storing elements in an array of references [11, §10.10]. This may cause an exception to happen, because Java arrays are *covariant* typed, i.e. *array of Point* is a subtype[1] of *array of Object* iff *Point* is a subtype of *Object*.

Programs which would otherwise cause runtime errors beyond these two are rejected either by the compiler or by the execution environment itself, which performs a type-inference integrity test on loaded code before it is executed. Regardless of these safeguards, it is clear that Java is not statically type safe – and it is even described in the Java specification. In fact most popular object-oriented programming languages have runtime type checks in one way or another, including C++, SIMULA, EIFFEL and BETA.

[1] Throughout this paper, we will assume that any type is by definition a subtype of itself. We will use the term proper subtype to express subtypes that exclude the type itself.

Exactly like C++, Java implements *overloading* and *no-variance* with respect to methods. This means that when a virtual method is overridden in a subclass, the overriding method must have exactly the same types of arguments and exactly the same return type. If the types of arguments are different, it is considered to be a completely different method which is said to be *overloading* the original method.

Some readers might find it interesting that Java also has the flavour of contravariance. Part of a method declaration in Java can be a `throws` clause, which designates the exceptions that the method may cause. An overriding method is allowed to declare the same set of exceptions as the method being overridden, or a subset thereof.

Java allows declaration of `final` methods and classes, designating such methods that cannot be overridden, and such classes which cannot be subtyped. This feature can be used to reduce the (performance) implications of subtype substitutability, and as a protection mechanism. In essence, `final` bindings can narrow the general *open world assumption*. This mechanism is much like Dylan's sealed classes [3] and also has similarities to final bindings in BETA [19].

3.3 Critique

In our opinion, Javas type system reflects a rather ad-hoc mix of the three desirable properties (covariance typing, full static typing and subtype substitutability), of which only subtype substitutability is available in a coherent fashion. Covariance is only available for arrays – methods are no-variance typed, while the language as a whole is actually not statically type safe. Since the language is not fully statically typed anyway, we will argue that covariance typing for methods may as well be introduced; while still preserving a high degree of static type safeness.

The absence of covariance typing in particular leads to a programming style in which many things are typed simply as `Object`, or some other abstract superclass, and values are then explicitly *casted down* when needed. The same programming style is generally used in C++, which has a type system with many similarities with Java's type system.

While there will always be a need to rediscover the full type of an object when storing such in some kind of collection, it is our opinion that there are many situations where Java's type system forces programmers to use excessive amounts of casts. Indeed it is ironic from a programmer point of view to have to write casts that will never cause type errors. Consider for instance the case when the programmer knows that a certain collection of objects only contain instances of a certain class, all the casts necessary to access objects in the collection would be superfluous.

3.3.1 Virtual Types and Java's Type System

With virtual types most of these superfluous casts that would normally be necessary do not have to be written explicitly by the programmer. Based on virtual type declarations, our compiler inserts casts automatically "behind the scenes."

These casts are actually needed in the code to allow the Java execution environment to perform it's type-inference based integrity check. The compiled Java code is then annotated with enough information to let the execution environment eliminate these superfluous casts again, thereby making the code run faster. If the Java execution environment is not aware of virtual type annotations, the program will simply run a little slower than if it is.

While virtual types will not guarantee programs to be type safe, it does allow more programs to be statically type safe. Cf. the discussion above, since Java is already not statically type safe and never will be, we will allow ourself to introduce covariance; effectively allowing an overriding method to specify more specific types for argument and return types.

Virtual types can also explain the covariant behaviour of Java arrays. Arrays can be thought of as a class with a virtual type describing the constraint on the elements; consider:

```
class Array {
  typedef T as Object;
  T insert (int index, T elem) { ... }
  T get (int index) { ... }
}
```

For any particular parameterization of the array type, say `String[]`, the `Array` class is specialized:

```
class StringArray {
  typedef T as String;
}
```

Which has the behaviour that any assignment into the array (using `insert`) is checked and can cause a type exception to happen.

4 Genericity and Inheritance

At present, the official Java language presents no way to express generic classes, i.e. classes which are polymorphic in one or more type variables, even though several proposals for such exists [22, 25]. Here we discuss some of the mechanisms that has already been deployed to obtain genericity in object oriented programming languages. A more encompassing review and critiqûe of genericity in various programming languages can be found in [27, 139ff].

4.1 Parameterized Classes

In his often quoted paper [20] Bertrand Meyer presents parameterized classes as a programming language mechanism in Eiffel which combines the benefits of polymorphism, as in ML, and those of inheritance. Several other programming languages, including C++ [8] and Sather [26] implement similar features. As we see it, there are several conceptual problems in parameterized classes.

To illustrate these problems, assume for a moment that Java does have parameterized classes, allowing declarations of the form:

```
class Map<Key,Elem> {
  void insertAt (Key k, Elem e) ...
  Elem elementAt(Key k) ...
  int count ();
}
```

Which defines a generic class `Map` with two *type parameters*, key and element. This generic class can then be used to create any kind of map, like a map from `Strings` to `Points` as illustrated here:

```
Map<String,Point> map1 = new Map<String,Point>();
```

The variable `map1` now refers to an instance of class `Map<String,Object>`, which is in turn an instance of the generic class `Map`. Because of this *instance of* relationship between classes and generic classes, there cannot be a straightforward subclass relationship between parameterized classes and regular classes, which seem desirable.

We consider it a conceptual problem that parameterized classes have to be *instantiated* in some sense to become "real classes," thus introducing another layer of abstraction in the model, somewhat like meta classes in CLOS and Smalltalk [10, 13].

Secondly, it is unclear what the relationships between parameterized class instantiations should be in these object models, especially if there is more than one type parameter. E.g. should `List<Point>` be a subtype of `List<Object>`, assuming `Point` is a subclass of `Object`? Getting back to our example, consider some other instances of the generic class `Map`:

```
Map<String,Object>
Map<Color,String>
Map<Point,Color>
```

And it is clear that only the first, i.e. class `Map<String,Object>` *may* have a sub-typing relationship to class `Map<String,Point>`.

Because of these conceptual problems, some programming languages either define explicitly *no* subtype relationship between such generic class instantiations as is the case in C++ or Pizza [25], or they define some kind of structural conformance semantics which is, in our opinion, only useful for one type parameter as is the case in both Eiffel and Sather, as well as the alternative proposal for parameterized types for Java by Myers, et. al. [22].

Like inheritance, conformance is yet another kind of mechanism which introduce types, and this will simply confuse the programmer. We find it compelling to think of inheritance itself as *the* mechanism for genericity. In some abstract sense, when a class is subclassed, it becomes less generic, while at the same time the original class acts as a "pattern" for the subclass. We use this similarity between inheritance and generic classes, implementing both through inheritance.

With our proposal the types that exist in a program are exactly those that are declared as classes, and the subtype relationship between these types is given explicitly in the inheritance hierarchy. Instead of creating another instantiation

of a conceptual "template class" the programmer will simply write another class. While this may at times cause the code to be slightly more verbose, it is conceptually much clearer than parameterized classes.

4.2 Recursive Class Types

Another class of problems related to genericity and inheritance arise when a class need to refer to itself. Consider for instance, an `equals` method in a class hierarchy of `Points` and `ColorPoints`:

```
class Point {
  int x, y;
  ...
  boolean equals (Point other) {
    return (x == other.x) && (y == other.y);
  }
}
```

Now, what should `equals` look like in a subclass, say `ColorPoint`? One solution might be to write two methods, one for `Points` and one for `ColorPoints`.

```
class ColorPoint extends Point {
  Color c;
  ...
  boolean equals (ColorPoint other) {
    return super.equals (other) && c.equals (other.c);
  }
  boolean equals (Point other) {
    throw new Error ("TypeError");
  }
}
```

...and while this is indeed a solution to this problem, it would be nice if this kind of "type error" could be checked by the compiler. With virtual types self-recursive types are trivially supported by declaring the argument of `equals` to be of the special `This` virtual type available to all classes, as it was already outlined in Section 2. Understanding recursive class types has been a subject of research in computer science for several years, further discussions can be found in [4, 5, 27]. Several programming languages include specific support for recursive class types, such as Sather [26] which has a special type named `SELF`, and Eiffel's `like Current` mechanism [21].

4.2.1 Recursive Classes in Design Patterns

Mutually recursive classes, which often occur in design patterns can easily be programmed. Consider in this example due to Erik Ernst, a simple implementation of the Observer pattern [9] in the following example, which also shows how virtual types can be used in interfaces (example due to Erik Ernst):

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

The usage of virtual types ensures that these classes work together, even in specializations of the presented classes. To use this design pattern in a particular context, the two classes would be subclassed "together," extending the various virtual types accordingly. A set of `Subject/Observer` classes for dealing with window events might look like this:

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

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

Following which any class can choose to implement the `WindowObserver` interface and in assurance that it only receives events that are at least `WindowEvents`, which are originating from a `WindowSubject` or a subclass thereof.

The BOPL language described in [27] explicitly supports mutually recursive class types through "automatic" generation of groups of classes that are statically safe, similar to how we manually generated classes for the example above.

5 Design

5.1 Terminology Considerations

Before we dive into the details of virtual types, we will discuss our choice of terminology a little. As described in the introduction, the spirit of Java is to only use mechanisms that are well known, and can be understood by all programmers. There is no point in introducing a mechanism which is too hard to understand.

Since Java's terminology and syntax is heavily inspired by C++, we have chosen to explicitly take this point of view in this presentation, and bring forth the `typedef` keyword. In C and C++, `typedef` introduces a simple type alias. Thus, coming from a C background, a programmer already knows the basic meaning of the keyword.

One might consider using `virtual typedef` then, to designate that the typedef can be extended in a fashion much like virtual methods. However, since *all* methods are by default virtual in Java, that distinction is not natural – why should typedef's have to be declared explicitly virtual when methods do not? In BETA a virtual type *definition* is syntactically distinguished from a virtual type *extension*. We chose not to make this distinction since Java doesn't syntactically distinguish methods that override other methods, from methods that do not override.

As a general name for the feature, we chose the name *virtual type* because we find it clearly descriptive: the name associates with typing, i.e. a virtual type really is a type; and because it associates with virtuality, i.e. a virtual type can be redefined in context of a subclass. We anticipate however, that our choice of syntax may lead to this feature being known as virtual typedefs, but that wouldn't be too bad.

5.2 Virtual Types Specifics

Now we introduce the full syntax for virtual type declarations, and the following sections will informally introduce the semantics of various aspects of virtual types. The first thing we describe is where a virtual type is allowed to appear. A *VirtualType* as defined in the following can be used anywhere a *ReferenceType* [11, §4.3] can be used:

ReferenceType:+
    VirtualType

VirtualType:
    TypeName
    `This`

Intuitively, this means that a virtual type can be used anywhere where a class or interface type could be used, e.g. to qualify instance variables, parameters, return values, etc.

5.3 Virtual Type Declarations

A virtual type declaration can have a *qualification* described by one or more classes or interfaces, a *name*, and optionally an explicit *binding*. A virtual type declaration has the following form:

VirtualTypeDeclaration:
    $Modifiers_{opt}$ `typedef` *Name*
        `as` *Qualification* $Binding_{opt}$ `;`

Qualification:
    ClassOrInterfaceType
    Qualification , *InterfaceType*

Binding:
    = *ClassType*

Modifiers: *one or more of*
    `final abstract`
    `public protected private`

Informally, the *Qualification* describes what you can assume about variables of type *Name*. The *Binding* is the class used when creating instances of the virtual type *Name*. The details of these is the subject of the following sections.

5.4 Virtual Type Qualifications

Any reference entity (variable, parameter, etc.) which is typed with the virtual type *Name* is allowed to refer to objects that are subtypes of all the types listed in the *Qualification* for the named type. The following defines the subtype relation $\subseteq$ between a type S and a list of types $[\![T_1, T_2, \ldots, T_n]\!]$, defined in terms of a subtype relation between singular types:

$$S \subseteq [\![T_1, T_2, \ldots, T_n]\!]$$
$$\Updownarrow$$
$$\forall t \in \{T_1, T_2, \ldots, T_n\} : S \subseteq t$$

Intuitively, this means that in order to be a subtype of a list of types, the type in question must be a subtype of each type in the list.

Further notice that the qualification does not need to list any classes, it is allowed to list only interfaces. In Java the special `Object` class is a super type of all other reference types including interfaces, so `Object` is implicitly inserted as the class in the qualification, if only interfaces are listed. For the same reason, a virtual type qualified only with `Object` is effectively an unconstrained virtual type.

As an example of a qualification listing multiple types consider the following declaration of a variable that can hold a reference to an object which is an instance of a class implementing both the `Encoding` and the `Decoding` interface.

```
typedef Archivable as Encoding, Decoding;
Archivable a;
```

Next consider this class `ArchivablePoint`, which implements the two interfaces `Encoding` and `Decoding`:

```
class ArchivablePoint implements Encoding, Decoding {
  ...
}
```

Since `ArchivablePoint` is a subtype of both `Encoding` and `Decoding`, a reference of type `ArchivablePoint` is allowed to be assigned to a reference of type `Archivable`:

```
a = new ArchivablePoint ();
```

As we shall see later however, this assignment has to be dynamically checked in the case where `Archivable` may be extended in a subclass.

A similar notion of "structural qualification" exists in Objective C [24], where any *variable* can be qualified by a class, and zero or more interfaces.[2] For instance, in Objective C, the following declares a variable `var` qualified by class `View`, which implements both the `Encoding` and `Decoding` interfaces.

```
View <Encoding,Decoding> *var;
```

Such qualifications are very useful because they allow arbitrary combinations of interfaces for a particular situation, without having to introduce a new class.

5.5 Extending Virtual Types

When a virtual type has been declared in context of a class or interface, it can be *extended* in subclasses (or sub-interfaces) thereof. Intuitively, extension of types is similar to the notion that a virtual method can be overridden in subclasses.

When a virtual type is extended, the qualification of the extended type must express a subtype of the qualification of the type being extended. Since qualifications can be lists of types, we need to define the subtype relationship between two lists of types:

$$[\![S_1, S_2, \ldots, S_m]\!] \subseteq [\![T_1, T_2, \ldots, T_n]\!]$$

$$\Updownarrow$$

$$\forall t \in \{T_1, T_2, \ldots, T_n\} : (\exists s \in \{S_1, S_2, \ldots, S_m\} : s \subseteq t)$$

In natural language this simply means that for each element t in the list describing the super type, there exists an element s in the list describing the subtype, such that t is a super type of s.

If the virtual type being extended is not declared `abstract` then it is allowed to be instantiated, as it will be discussed further in section 5.7. If this is the case then the constructors of the qualification class are exposed, so it must also be asserted that these are available in the more specific type. Since constructors are not inherited in Java, it is a further requirement that the same constructors are available for the qualification class in the extending type as in the type being extended.

The following is an example of using multiple types in the qualification of a virtual inspired by David Shang's cow example [30].

[2] Java's notion of interfaces originates from Objective C, where they are called formal protocols, as opposed to Smalltalk's informal notion of protocols. The mechanism was fostered by Steve Naroff at NeXT in 1991, and first released in NeXTSTEP 3.0 [23].

```
class Animal {
  typedef Edible as Food, Drink;
  eat (Edible e) { ... }
}

class Cow extends Animal {
  typedef Edible as VegetarianFood, Water;
  eat (Edible e) { ... }
}
```

These rules for extension of virtual types express subtype substitutability, i.e. the general notion that a subclass is more specific than it's superclass, while at the same time being "upwards compatible." A consequence of this rule is the introduction of *covariant* method typing, since we allow method arguments to be qualified by a virtual type.

If a virtual type is declared `final`, then it cannot be extended in classes or interfaces which inherit the given virtual type. It is trivially true that any member of a `final` class is automatically `final` as well, so a virtual type may also implicitly be `final` by appearing in a `final` class.

5.6 Virtual Type Casting

Virtual types can also be used in a dynamic cast expression. When either explicitly or implicitly introducing a cast to a virtual type of the form:

var = (*Name*) *expr* ;

then *expr* must be either `null`, or a reference to an object satisfying the qualification of *Name* as declared in the *dynamic* class of `this`.

Should the qualification for a virtual type be broken, then the runtime system will throw a `java.lang.VirtualTypeCastException` runtime exception. This is similar to the exceptions that may happen when storing elements in an array, or when casting values of reference type as described in Section 3.2.

A dynamic cast like the one above is sometimes inserted implicitly when passing an argument to a function where the parameter is declared to be of a virtual type. Specifically, it is not needed if the value being passed as an argument is already qualified by the virtual type, or if the virtual type is declared `final` in the given context.

5.7 Virtual Type Bindings and Instantiation

The effect of instantiating a virtual type is to create an instance of its binding. An instance of a virtual type is created using the following syntax:

var = *obj*.`new` *Name*() ;

where *obj* is a reference to an object declaring the virtual type designated *Name*. The "*obj*." prefix can be omitted if *Name* is accessible directly in the enclosing context (via `this`), as for other member accesses.

The optional *Binding* form in the declaration syntax allows an explicit specification of the class to be the binding of the virtual type. If no explicit binding is declared, and the *Qualification* lists exactly one element which is a *ClassType*, then the binding defaults to that class, i.e. "`typedef` *Name* `as` *C*" is shorthand for "`typedef` *Name* `as` *C* = *C*" for any class *C*.

Since constructors are not inherited in Java, it is not trivially true that the binding class has all the constructors available for the qualification class. Thus it is a further requirement that the binding must have *at least* the same constructors as the class in the qualification; a constraint that can easily be checked at compile time.

If a virtual type is explicitly declared `abstract` then it cannot be instantiated, similar to abstract classes. Alternatively, if a virtual type is *not* declared `abstract`, then it is always allowed to be instantiated. From this basic notion of abstractness we then derive rules for the cases where virtual type needs to have an explicit binding.

Virtual types declared in interfaces are never allowed to have bindings. This is similar to the notion that methods in an interface cannot have a body. When a class inherits an interface which declares a virtual type and that virtual type is non-abstract (i.e. is allowed to be instantiated), then the class must "implement" the virtual type by explicitly declaring the virtual type with a binding.

The constructors available for instantiating a virtual type are limited to those that are available for the class type appearing in the qualification, or if no class type is in the qualification list only the default constructor is available. This means that for all intent and purposes, the binding of a virtual type is not visible, only the qualification is.

As for any object with virtual type, the class appearing in the *Binding* form has to be a subtype of the virtual type, so the binding class also has to satisfy the qualification.

5.7.1 Instantiation of the special type `This`

Because the `This` virtual type is uniformly available for all classes, it's use for instance creation is restricted to only use the default constructor. This means that the virtual type `This` effectively adopts the same access protection level as the default constructor. The restriction that only the default constructor is available via `This` is not a problem in practice, because it is easy to decouple the initialization from allocation, writing a virtual `init` method which can take whatever arguments.

5.8 An Example

Below we bring a larger example using virtual types, defining a class `Ring` and related classes. This example was introduced in [20], and has been used in the literature to illustrate mechanisms for genericity [18, 19, 27]. This example also illustrates how the `This` virtual type can be used to statically type self referential classes.

```
abstract class Ring
{
  public Ring() { zero(); }
  void plus (This other);
  void zero ();
  void unity ();
}

class ComplexRing extends Ring
{
  double i, r;
  void plus (This other) {
    i += other.i; r += other.r;
  }
  void zero () {
    i = 0.0; r = 0.0;
  }
  void unity () {
    i = 0.0; r = 1.0;
  }
}
```

```
class VectorRing extends Ring
{
  typedef ElementType as Ring;
  ElementType e[];
  VectorRing (int size) {
    e = new ElementType[size];
    for (int i = 0; i < size; i++)
      e[i] = new ElementType();
  }
  void plus (This other) {
    for (int i = 0; I < e.length; i++)
      e[i].plus (other.e[i]);
  }
  void zero () {
    for (int i = 0; I < e.length; i++)
      e[i].zero();
  }
}

class ComplexVector extends VectorRing
{
  typedef ElementType as ComplexRing;
  ComplexVector (int size) {
    super(size);
  }
}
```

6 Implementation

As for implementation, there are three major implications: dynamically checking that an object qualifies for a given virtual type; selecting where to insert such checks; and creating instances of virtual types. In the following we will outline how these are implemented. A full specification is beyond the scope of this paper.

Of primary concern in our design is to make virtual types integrate closely with the Java programming language as it is. In the implementation we have therefore explicitly chosen to restrict ourselves such as to:

- Perform translation into pure Java or translation directly to the Java byte code format.
- Make the translation in such a way that existing classes will not have to be recompiled to be used from a class with virtual types.
- Let the resulting generated classes be usable via a compiler that does not understand virtual types.

Since Java byte code format is tightly coupled with the language itself, there is technically very little difference between generating Java code and generating

byte code. In this presentation the translation is presented as a source code mapping.

6.1 Substitution of Virtual Types with Real Types

For the compiler, the very first *ClassOrInterfaceType* appearing in the qualification of a virtual type definition is special. This type is substituted for every usage of the virtual type as the qualification of some other entity, such as an argument or a variable. This most general specific type of the virtual is also used for all applications of extension of this virtual type. Consider the following example:

```
class Vector {
  typedef T as Object;
  insert (T elem) { ... }
}
class PointVector extends Vector {
  typedef T as Point;
  insert (T elem) { ... }
}
```

The `Object` is substituted for the all the occurrences of `T` in the code, and when a method is overridden, an explicit cast is inserted yielding:

```
class Vector {
  insert (Object elem) { ... }
}
class PointVector extends Vector {
  typedef T as Point;
  insert (Object elem$0) {
    Point elem = (Point)elem$0; ...
  }
}
```

This imposes a non-trivial impact on the performance of the system, and since the compiler can guarantee that this cast will never fail, it can augment the generated byte code with extra information so the virtual machine may eliminate the redundant cast.

6.2 Dynamic Cast to Virtual Types

A dynamic cast to a virtual type is implemented by a virtual method call. For each *VirtualTypeDeclaration*, the compiler will generate a virtual method named "`cast$`*Name*", taking as argument and returning an object qualified by the first element in the qualification for that virtual type. This method will check that the object conforms to the given qualification by attempting to cast the incoming object to all the required types. For instance, for the virtual type declaration:

```
class Vector {
  typedef T as Observer;
  ...
  { ...; T var = (T)expr; ... }
}
```

The compiler will generate the following, "`return (Observer)o`" being the heart of the cast operation.

```
class Vector {
  Object cast$T (Object o) {
   try { return (Observer)o; }
   catch (ClassCastException e) {
    throw new
      VirtualTypeCastException (...);
   }
   ...
  { ...; Object var = cast$T(expr); ... }
}
```

If the cast fails the Java runtime will generate a `ClassCastException`, which in turn is translated to a `VirtualTypeCastException` by the code in the `catch` clause.

A virtual type declared in an interface will generate an abstract declaration for the according cast operation, thus imposing a requirement on implementors of that interface to actually implement the cast operation.

6.2.1 Compiling casts to `This`

The compiler automatically adds a virtual type named `This` to all classes, similar to how the `Ring` example used the class. For this, code of the form:

```
T var = (This)obj;
```

is translated to:

```
T var;
if (this.getClass().isInstance(obj))
  var = (T)obj;
else
  throw new VirtualTypeCastException (...)
```

Using this special implementation has the advantage that existing classes will not need to be recompiled in order to adopt the special `This` virtual type, and subclasses of the class containing the compiled code will automatically have the correct behaviour even if they are compiled with a compiler not supporting virtual types.

6.3 Insertion of Virtual Casts

Whenever a call is made to a method with arguments of virtual type or when a value is assigned to a variable of virtual type, the assigned value may have to be checked dynamically. However, for a large number of such situations, the check can be eliminated since it can be determined statically that the type is already right.

Because these casts are not necessary everywhere, the check is performed at the call site, rather than inside all methods taking virtual type arguments.

Consider the following example of making a call to `insert` of the `Vector` class, where we need to insert a dynamic check:

```
class Vector {
  typedef T as Object;
  void insert (T elem) { ... }
}
...
Vector l; ... l.insert (expr);
```

For which we generate the equivalent of:

```
Vector l; ... l.insert (l.cast$T(expr));
```

Because the variable `l` may be referring to a subclass of `Vector`, which has a more stringent qualification than visible directly by inspecting class `Vector`.

Of particular interest is all the situations where these checks are not needed. This is the case whenever the assigned value can be determined to be the same actual virtual type as required, as opposed to the declared virtual type as seen from the usage site. This is the case in the following two situations:

- When calling a method via `this`, with an argument already typed as the same virtual type T. In this situation both the formal parameter and the given value will have the type `this.`T.
- When the virtual type as seen from the usage site is `final`, either because it was declared so explicitly or because the class it appears in is declared `final`.

To illustrate the first case above, assume we extend the `Vector` class above with a method `insertIfAbsent`:

```
class Vector {
  typedef T as Object;
  void    insert (T elem) { ... }
  boolean includes (T elem) { ... }
  void    insertIfAbsent (T elem) {
    if (!this.includes (elem))
      this.insert (elem);
  }
}
```

In the translation for `insertIfAbsent`, the type of `elem` does not have to be checked for the two calls to `includes` and `insert`, because it has already been verified before entering `insertIfAbsent`.

The second case above is trivial to see, because it simply describes the situation where a virtual type is made non-virtual by declaring it `final`. For performance reasons it is often a good idea to make such classes `final`, which are not designed to be specialized, because this will also reduce the overhead of any regular method invocations to the cost of a regular function call.

6.4 Instance Creation of Virtual Types

As described earlier, if a virtual type is not explicitly declared `abstract` it must have a binding, i.e. be capable of being allocated. To facilitate this, any such virtual type will generate a set of methods named "`new$`*Name*", one for each constructor of the class listed in the virtual type qualification. If no class is listed in the qualification, only the default constructor is generated. Here is an example like above, except here we give `T` a binding to allow it to be instantiated.

```
class Vector {
  typedef T as Observer = WindowObserver;
  ...
  { ...; T var = new T(); ... }
}
```

The compiler will generate the following.

```
class Vector {
  Object new$T () {
    return new WindowObserver ();
  }
  ...
  { ...; Object var = new$T(); ... }
}
```

If the type `T` is extended in a subclass, then the method generated for the extension will override the method declared here.

6.4.1 Compiling allocation of `This`

Similar to the case for casts above, allocation of the virtual type `This` is compiled especially. For this, code of the form:

```
T var = new This ();
```

is translated to:

```
T var = this.getClass().newInstance();
```

Using this special implementation has the advantage that existing classes will not need to be recompiled in order to adopt the special `This` virtual type.

6.5 Performance Impact

One place where virtual types introduce a non-trivial overhead is when calling a method with virtual typed arguments. Consider for instance the `insert` method in the `PointVector`, which has been used throughout this article. Here is an example using that method:

```
{ PointVector pv; Point pnt; ...
  pv.insert (pnt); ... }
```

Because of subtype substitutability, pv may refer to an instance of a subclass of `PointVector`, which may again *extend* the virtual type of the declared argument. In this case, our implementation inserts an extra call, asking pv to assert that the argument does indeed match the constraint it imposes on `T`. This generates the equivalent of the following:

```
{ PointVector pv; Point pnt; ...
  pv.insert (pv.cast$T(pnt)); ... }
```

Effectively making a virtual method call become two virtual method calls. While this is indeed a problem, we have made our implementation in such a way that a compiler using customization may inline one of the virtual calls effectively reducing this overhead to a dynamic cast. With customization, calls to `this` can all be inlined, so we introduce a new method in `Vector` which performs the `cast`-checks, and calls to `insert`. Calls to `insert` are then replaced with this call when needed. Because virtual `cast$T` method is then overloaded in `PointVector`, the check will do the right thing. The following code snippets illustrate how the check is rewritten. The two calls appearing in `check$insert` can both be inlined in a customizing compiler.

```
class Vector {
  Object cast$T(Object o){ return o;}
  void check$insert(Object o) {
    this.insert(this.cast$T (o));
  }
  void insert(Object o) { ... }
}

class PointVector extends Vector {
  Object cast$T(Object o) {
     try { return (Point)o; } ...
  }
}
```

It is not necessary to make this check when the type of argument can be determined to be sound, such as when calling the insert method with an argument already qualified by `T`, or when the type of `T` is final.

Much work has already been done to implement such compilers supporting customization as part of the SELF project [2, 31]. One such sophisticated Java

virtual machine is described in [1]. Another similar Java virtual machine has been developed by Animorphic Systems which was recently acquired by JavaSoft.

If Java were to support *type exact variables*, i.e. variables that can only hold references to instances of a particular class but not its subclasses, then this qualification check could be eliminated because the exact qualification of a virtual type in objects referred to would be known. Sather is one of the languages that support type exact variables. In lieu of this, Java's `final` classes can be used to simulate type exact variables, since variables qualified by a class declared `final` can only refer to instances of that class. If `PointVector` was defined as the following, calling methods on this would be no more expensive than it is with todays collection libraries.

```
final class PointVector
    extends Vector
{
  typedef T as Point;
}
```

Similarly, this extra overhead can be eliminated by declaring the extended virtual type declaration final, like in the following:

```
class PointVector extends Vector {
  final typedef T as Point;
}
```

That way, it would still not cost extra to call the methods of `PointVector` with virtual type arguments, *and* the class `PointVector` can be subclassed.

6.6 The Cost of a Cast

Since the overhead we impose effectively is a matter of some extra casts, we have been trying to estimate the cost of dynamic casts in general. In an effort to estimate the cost of a regular dynamic cast in Java, we tried to run the same test program Myers et. al. used in [22] to determine that:

> "For a simple collection class, avoiding the runtime casts from `Object` reduced the run times by up to 17%..."—*Myers, Bank, and Liskov*

Their performance figures also compare the cost of "cast from `Object`" to "hard coded types", and they state that using hard coded types, i.e. not having to cast, is as much as 21% faster. We were not able to reproduce their results. For the following program, which resembles that in the paper of Myers et. al., we observed only on the order of 1-5% slowdown, with variations for various hardware and virtual machines. We tried running it using Sun's JDK on various SPARC platforms; as well as running it on a 486DX100 using the Microsoft JIT-based environment.

```
Vector v = new Vector ();
v.addElement (new Point (0, 0));
```

```
for (int j = 0; j < 100000; j++) {
  Object t = v.elementAt (0);
  t.equals (t);
}
for (int j = 0; j < 100000; j++) {
  Point t = (Point)v.elementAt (0);
  t.equals (t);
}
```

Since developing these results we have obtained a copy of the actual code used by Myers et. al. in their paper, which we have included below. While they state that their example is based on a "simple collection class" it is rather based on an "accessor method" for a single member variable in a tight loop, thus emphasizing the impact of a the cast. Given this, we still believe that our measurements presents a more realistic picture.

```
class CellElement {
  private Element t;
  public CellElement(Element t) {
    this.t = t;
  }
  public Element get() { return t; }
}

CellElement c = new CellElement();
c.add(new Element());
for (i = 0; i < 1000000; i++) {
  Element t = c.get();
  t.do_method();
}
```

In addition to replicating the test case used by Myers et. al we tried to run a performance test of the code generated for the `Ring` example above, comparing it to a "hand written" version which had no explicit casts. With that test case we saw a performance degradation on the order of one to two percent, depending on which configuration it was tested in. We believe that the performance of an average program is no worse than the `Ring` code in the example above.

While we have only performed very limited performance testing, we believe our results are much more realistic than those of Myers et. al. In order to obtain better figures for the run time performance impact of virtual types, one would have to write a large program twice, both with and without the feature, a job which has been beyond the scope of this project so far. Based on admittedly very limited tests, we see a strong indication that the performance overhead is in the order of one to two percent or less for any realistic application.

7 Discussion

While the previous sections presents a specific suggestion for how to include virtual types in Java which is complete in itself, we will here discuss some possible alternatives to our design ad identify some of the open issues. Finally, we will discuss related and future work.

7.1 Variations

Several reviewers have pointed out that it would be more in Java's spirit not to have the compiler insert the dynamic cast for method arguments of virtual type. Inserting these casts automatically is how BETA does it. Rather, they would like to require the programmer to insert explicit casts, so the invariant can be maintained, that run time errors can only happen in the two situations listed in Section 3.2, i.e., only at explicit casts and array-store operations.

This would indeed be very useful, but it would require significant changes to the current Java grammar. For instance, the syntactic category *VirtualType* would have to not only include "*TypeName*," but also "*AmbiguousName . TypeName*" [11, §6.5], so that a virtual type can be accessed using dot notion for casts, such as in:

```
void m (PointVector pv) {
  Point p = new Point(2, 2);
  pv.insert ((pv.ElemType)p);
}
```

Which would then be generating the following code:

```
void m (PointVector pv) {
  Point p = new Point(2, 2);
  pv.insert ( pv.cast$ElemType(p) );
}
```

Secondly, the compiler would have to be able to decide that the value of pv in the example above does not change between the two applications of that variable. If the value of pv would change between the cast and the call to `insert`, then the cast is no good, since it may have been replaced by some other subclass of `PointVector`. This assertion is trivially true for constant, i.e., `final` declared, fields, variables and parameters [11, §8.3.1.2], or the special variable `this` which is always constant. One possible restriction could thus be to only allow remote virtual qualifications (and casts) via `final` variables or fields.

In our design of virtual types we have decided not to include remote casts because of these complications, and thus virtual types are only accessible implicitly via `this`. On the other hand, the dynamic check is only used when actually needed, as outlined in the implementation section it can be eliminated when calling a method on `this`, when invoking a method on a `final` class, or if the virtual type in question is declared `final`.

A major complication in the design is the fact that constructors are not inherited. One idea we have been playing around with was to introduce a kind

of constructor which is automatically replicated in all subclasses unless explicitly redefined.

Another issue which we will not cover in detail in this paper is how to eliminate the "superfluous casts" that are inserted as part of the code transformations implementing virtual types. Intuitively, to make this possible the verifier will need to know about virtual types. To enable this, the compiler must add extra annotations to the *.class* file replicating the declaration of virtual types, and all their applications in method signatures and instance variables. By examining these in the same way the compiler did, the verifier can see "why" the compiler chose to insert certain casts, and thus it can also remove them again if they can be determined to be safe.

One of the weaknesses of our present design is that primitive types, such as `int` and `long`, cannot be used for qualification, and thus parameterization by simple types is not possible. While this is indeed a problem, it is based on an inherent notion in Java, that such simple types are *not* objects. Collections of simple objects can be made using the wrapper classes `java.lang.Integer`, `java.lang.Long`, etc.

Another weaknes is the fact that our support for recursive class types via `This` is limited to self-recursive types. It is harder to to support mutally recusrive classes that support being subclassed, such as is often the case for design patterns. To do this the programmer has to emulate the equivalent of `This` for the recursive types and extend these appropriately in subclasses. The BOPL programming language [27] solves this problem in a clean and consistent, albeit very expensive, fashion.

7.2 Related Work

In their paper [22], Myers et. al. suggest a mechanism for implementing constrained genericity through parameterized classes in Java, where constraints are based on *where* clauses as in CLU [15]. Another recent paper by Odersky et. al. is [25] which describes the design of Pizza, implementing of F-bounded parametric polymorphism for Java in a fashion very similar to Myers et. al.

While both of these are very well documented indeed, they still have the conceptual problems with parameterized classes discussed earlier. In addition, since in [22] *where* clauses are based on conformance rather than declared relationships between types, a class may accidentally conform to (match) the where clause without the programmers intent. This introduces another class of "semantic type error" illustrated with this example due to Boris Magnusson: Consider a clause of a graphics related method requiring a `draw` method to be present at objects it accepts as arguments. Now imagine handing an instance of `GunMan` to this method.

One of the strengths of both their designs is that they allow parameterization by simple types such as `int` and `float`. However, their implementation it limited to 32-bit entities (thus excluding `long` and `double`), and it is tied to the fact that they provide their own implementation of the Java VM. In their paper they briefly describe how parameterization can be implemented using only the virtual

machine, but forget to mention that it would also disallow parameterization with primitive types.

If at all possible, one should strive not to require changes to the Java Virtual Machine when designing changes to Java. There is already several dozen implementations of the Java Virtual Machine, so adding new virtual machine instructions is currently impractical, if not impossible. Myers et. al. argue that casts are too expensive to allow implementation of covariance typing without changes to the virtual machine. We believe this is wrong. Their position is supported with performance measurementns we believe to be carrying very little value, as already described in the section on performance above.

7.3 Future Work

We are currently investigating how the scheme presented herein could be generalized, so as to be usable for other languages with compilers targeted at the Java Virtual Machine. Since virtual types subsumes e.g. EIFFEL-style parameterized classes, and because it is very similar to generics in ADA it is concievable to that we can make one mechanism that can support such languages.

Another interesting feature would be to introduce *type exact variables*, i.e. variables that do not allow subtype substitutability. By using such variables, typechecking of covariance can often be eliminated so more programs can be statically typechecked. Sather and BETA are examples of languages which already implement type exact variables.

8 Conclusion

We have presented *virtual types*, a programming language mechanism known from BETA, as a means to provide the functionality of parameterized classes in the Java programming language. Virtual types can also be used to effectively express recursive class-types, structures which often occur in design patterns.

The real advantage of virtual types over "traditional" parameterized types is that virtual types provide a simple conceptual model for providing generic classes which very intuitive because the subtype relationship between all class-types in a given program are given directly by the inheritance hierachy. Other programming languages typically provide a mixture of typing features to obtain the same level of functionality, which may be confusing to the programmer.

Virtual types allow covariance methods, which is very useful and intuitive, in a style that can be statically type checked in many situations. When calling covariant final methods, the arguments can always be statically checked.

We have outlined our implementation of the virtual types mechanism. This is descibed as a transformation into the core Java programming language itself. As such, our translation does not require any particular runtime support, so the compiled programs can run in any Java execution environment.

Our implementation does impose some overhead. For passing arguments to methods which has formals declared of virtual type, the overhead is an extra

virtual method call. In most situations, a execution environment with sophisticated inlining can eliminate this overhead. For other cases, the overhead is determined to be neglectable.

After having worked out all the details in this paper, it is clear that the relative complexity of Java itself makes it hard introduce changes to Java, which are consistent with Java and at the same time easy to explain. While Java at first glance seems like a neat little language, The Java Language Specification [11] is more than 800 pages, and it is an unsurmountable task to know it all.

Acknowledgements

The author would like to present a special thank to BILL JOY and JOHN ROSE for their insightful comments and explanations of the finer points of Java. The author has also greatly benefitted from discussions, comments and encouragement from the following people: OLE LEHRMANN MADSEN, MADS TORGERSEN, GORDIE FREEDMAN, the anonymous reviewers, and last but not least my co-students who helped proof read the paper.

References

[1] AGESEN, O. Design and implementation of Pep, a Java just-in-time translator. To appear.

[2] AGESEN, O., AND HÖLZLE, U. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of OOPSLA'95* (1995), ACM Press, pp. 91–107.

[3] APPLE COMPUTER, EASTERN RESEARCH AND TECHNOLOGY. *Dylan: An object-oriented dynamic language*, 1st ed. Cambridge, MA, April 1992.

[4] CANNING, P., COOK, W., HILL, W., AND OLTHOFF, W. F-bounded qualification for object-oriented programming. In *ACM Conference on Functional Programming and Computer Architecture* (1989), ACM Press.

[5] CANNING, P., COOK, W., HILL, W., AND OLTHOFF, W. Interfaces for strongly-typed object-oriented programming. In *Proceedings of OOPSLA'89* (1989), SIGPLAN, ACM Press.

[6] DAHL, O. J., AND NYGAARD, K. Simula, an algol-based simulation language. *Communications of the ACM 9*, 9 (1966), 671–678.

[7] ELECTRONIC COMMUNITIES. The E Whitepaper. from http://www.communities.com/e, 1996.

[8] ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[9] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Abstraction and Reuse of Object-Oriented Designs.* Addison-Wesley, 1994.

[10] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80 – The Language.* Addison-Wesley, 1989.

[11] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* Addison-Wesley, 1996.

[12] JAVASOFT. Inner Classes in Java 1.1. from http://www.javasoft.com, 1996.

[13] KICZALES, G., DE RIVIÈRES, J., AND BOBROW, D. G. *The Art of the Meta Object Protocol.* MIT Press, 1991.

[14] KRISTENSEN, B. B., MADSEN, O. L., MØLLER-PEDERSEN, B., AND NYGAARD, K. Abstraction mechanisms in the BETA programming language. In *Proceedings of POPL'83* (Austin, TX, 1983).

[15] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction Mechanisms in CLU. *Communications of the ACM 20*, 8 (August 1977).

[16] MADSEN, O. L. Open issues in object-oriented programming–a scandinavian perspective. *Software–Practice and Experience 25*, S4 (December 1995).

[17] MADSEN, O. L., MAGNUSSON, B., AND MØLLER-PEDERSEN, B. Strong typing of object-oriented languages revisited. In *Proceedings of OOPSLA'90* (Ottawa, Canada, 1990), SIGPLAN, ACM Press.

[18] MADSEN, O. L., AND MØLLER-PEDERSEN, B. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA'89* (1989), SIGPLAN, ACM Press.

[19] MADSEN, O. L., MØLLER-PEDERSEN, B., AND NYGAARD, K. *Object-Oriented Programming in the* BETA *Programming Language.* Addison-Wesley, 1993.

[20] MEYER, B. Genericity versus Inheritance. In *Proceedings of OOPSLA'86* (1986).

[21] MEYER, B. *Object-Oriented Software Construction.* Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[22] MYERS, A., BANK, J., AND LISKOV, B. Parameterized types for Java. In *Proceedings of POPL'97* (1997).

[23] NAROFF, S. Personal communication. 1993–1996.

[24] NEXT COMPUTER, INC. *Object Oriented Programming and the Objective C Language.* Redwood City, CA, April 1993.

[25] ODERSKY, M., AND WADLER, P. Pizza into Java: Translating theory into practice. In *Proceedings of POPL'97* (Paris, 1997), ACM Press.

[26] OMOHUNDRO, S. The Sather Programming Language. *Dr. Dobb's Journal 18*, 11 (October 1993).

[27] PALSBERG, J., AND SCHWARTSBACH, M. I. *Object-Oriented Type Systems.* Addison-Wesley, 1993.

[28] SCHWARTSBACH, M. I. Object-oriented type systems: Principles and applications. from http://www.daimi.aau.dk/~mis, 1996.

[29] SEIDEWITZ, E. Genericity Versus Inheritance Reconsidered: Self-Reference Using Generices. In *Proceedings of OOPSLA'94* (1994).

[30] SHANG, D. Are cows animals? *Object Currents 1*, 1 (January 1996). http://www.sigs.com/objectcurrents/.

[31] UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HÖLZLE, U. Object, message and performance: How they coexist in SELF. *IEEE Computer 25*, 10 (October 1992).

[32] ZDONIK, S., AND MAIER, D. *Readings in Object Oriented Databases.* Morgan Kaufmann, 1990, ch. 1: Fundamentals of Object-Oriented Databases, pp. 1–32.