Interfaces for Nested Classes

Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
yannis@cc.gatech.edu

Abstract

Class nesting has emerged as a promising modularization mechanism in class-based object oriented languages. Several recent research results employ class nesting to concisely express relationships among multiple classes. Nevertheless, the issue of type system support for nested classes has not been addressed in previous work. In this paper, we discuss the problem in the context of Java and the Java "interface" mechanism. Interfaces are a way to specify explicit types for class definitions. Current Java interfaces cannot express constraints for nested classes. We will show that this is an important omission. An extension to Java is proposed that solves the problem and has two desirable characteristics: it does not change the semantics of existing Java programs and it does not introduce new keywords into the language. Additionally, we give advanced examples to support the general claim that type support for nested classes is useful for future OO languages and language extensions.

1 Introduction

Modern class-based object-oriented languages (e.g., C++, Java) support class nesting as a way of structuring code. The idiom of class nesting is particularly promising, as classes can serve as small-scale modules. Several recent research publications [5][15][21] have used class nesting to elegantly express relationships among multiple classes. Nevertheless, type system support for class nesting is rarely provided and has not been addressed as a research issue. The need for such type system support is quite apparent. For instance, Bruce, Odersky, and Wadler explicitly state in [5]: *"The intention is that any implementation of the 'outer' interface [...] must provide implementations of the 'inner' interfaces."* This requirement only appears on paper, however. The language of Bruce et al's example (Java) does not enforce it automatically.

In this paper, we discuss ways to enforce type constraints for nested classes in the context of Java. In particular, we present an extension of the Java interface mechanism to handle constraints on nested classes. The nature of the language extension requires some thought, as some obvious candidate syntax (nesting interfaces inside interfaces) is already allowed by Java and does not have the desired semantics. Our proposed solution does not change the semantics of existing Java programs (which are oblivious to our extension) and does not introduce new keywords in the language.

Apart from describing a specific extension mechanism, this paper attempts to make a more general case for the value of types for nested classes. This is an interesting issue in the design of future OO languages or language extensions. We present specific examples from recent research work in the literature. For instance, like any type mechanism, interfaces for nested classes are most valuable in the presence of *variables* over a given interface. Classes in Java are not first-class objects (e.g., the type system does not support variables ranging over classes). Nevertheless, class functors can be developed by combining inheritance and parameterization. Several parameterization mechanisms for Java have been proposed (e.g., [1], [4], [6], [11], [18]). We show that our extended interfaces are very useful in conjunction with some of the parameterization mechanisms, for expressing constraints on template parameters.

2 Background and Motivation

2.1 Interfaces

Interface specifications in object-oriented (OO) languages are a way to explicitly express type constraints on classes. Inter-

faces, by nature, form an incomplete constraint language. Even though some properties are easy to express (for instance, "class `A` should support a method `foo` that takes no argument and returns an object of class `B`") others are not expressible (for instance, "class `A` should support a method named `foo`"). The restrictions are usually imposed because of technical limitations (e.g., simplified parsing) and lack of significant need in everyday programming. Nevertheless, we argue that there is an important omission from the interface constraint language of Java. This has to do with expressing properties for nested classes. In particular, there is no way to constrain a class with respect to the nested classes that it must contain.

Interfaces are used to specify explicit type signatures for class members. Consider the example of a Java `interface` declaration:

```
interface Foo1 {
  Foo1     meth1 ();
  boolean  meth2 (Foo1 foo);
}
```

Any (concrete) Java class that conforms to this interface has to define two methods `meth1` and `meth2` with the exact type signatures specified in the interface. Conformance is declared using the `implements` keyword. For instance:

```
class Baz implements Foo1 {
  public Foo1 meth1() { return new Baz(); }
  public boolean meth2(Foo1 foo) { return true; }
}
```

The definition of `Baz` is legal because it supports both methods prescribed by the interface (i.e., it defines both methods with *identical* signatures to the interface prototypes).[1] It should be clear from this example that interface specifications are simply type constraints on class definitions.

2.2 Nested Classes

Nested classes are a powerful mechanism for integrating some of the benefits of block-structured programming in object-oriented programming languages. Nested classes in Java [7] behave in many respects like other class members (methods and member variables): they are inherited by subclasses, they have the same access control specifiers (e.g., `public`, `private`), and the outer class acts as a namespace for scoping purposes. (We will ignore the distinction between Java nested top-level classes and inner classes, as it is not pertinent to our discussion.) Unlike other class members, however, nested classes are not supported by the interface specification mechanism. In particular, there is a kind of constraint which would be particularly beneficial if expressible using interfaces. Its general form is:

- the constrained class should contain publicly visible nested classes that conform to selected predefined interfaces.

Note that this constraint refers to "interface conformance". The definition of conformance includes this constraint, hence, the constraint specification is recursive and can be nested arbitrarily. (For instance, we could specify the property "class `A` should contain nested class `B` which contains nested classes `C` and `D` conforming to interfaces `I` and `J`, respectively".)

Next, we will describe informally a small set of changes to the Java syntax (as well as the corresponding extensions to the semantics) to support such constraints. It should be straightforward to adapt the main ideas of our extensions to different concrete syntax (even in different languages, e.g., C++).

3 Interfaces for Nested Classes

3.1 A Failed Attempt

We will explore the design space for extending Java to support type constraints for nested classes. There is an obvious candidate syntactic form for expressing interface conformance in Java: using nested interfaces to signify that a client of the outer interface should include nested classes implementing the inner interfaces. Unfortunately, this option will not work because it would change the semantics of existing Java programs, since interface nesting, as it is at present in Java, has no semantic consequences on classes that implement the interface—it only delimits the scope of the nested interface declaration. Furthermore, the existing semantics for this syntax makes sense for fairly realistic examples, as we will argue. Thus, it

1. Java is *non-variant* with respect to method signatures in superclasses and interfaces. The language is *co-variant* with respect to arrays: an array of subclass instances can be used in place of an array of superclass instances [18].

is not a good idea to use this approach, not only in the context of Java, but also for future languages.

To see this, consider the consequences of nesting an interface inside another interface (meaning that clients of the outer interface should have a nested class implementing the inner interface). This would break code that uses interface nesting with no implications for the members of clients. Furthermore, the reasons to use such code with the existing Java semantics can be fairly legitimate. For instance, we show below a skeletal data structure interface:

```
interface DS {
  interface IfElement {
    ... // whatever
  }

  void      insert(IfElement element);
  boolean   find(IfElement element);
}
```

It is fairly natural to nest interface `IfElement` inside interface `DS` (and this syntax is legal Java). `IfElement` will specify some fundamental operations that any member of a `DS` data structure should support. On the other hand, it is unnatural to require by default that clients who implement `DS` must also contain a nested class that implements `DS.IfElement`. For instance, a given client that implements `DS` may support multiple types of elements, each of which just declares that it implements `DS.IfElement`. The client of `DS` may not even know all the different types of elements it supports, so nesting their definitions inside the `DS` client definition is not feasible.

3.2 Our Proposed Solution

We can express type constraints for nested classes by allowing *class prototypes* to be nested inside interfaces. By "class prototype" we mean a class declaration with no class body. The current syntax for class declarations is:

`ClassDecl: ClassModifiers(opt)` **`class`** `Identifier Super(opt) Interfaces(opt) ClassBody`

Our proposed syntax for class prototypes is:

`PrototypeDeclaration: ClassModifiers(opt)` **`class`** `Identifier Interfaces(opt)`

with the restriction that prototypes can only appear nested inside an interface. This is analogous to the current Java syntax for method prototypes in interfaces. The semantics for this extension is straightforward: we specify that *a class prototype nested inside an interface declaration means that classes conforming to the interface should have a publicly available nested class conforming to the prototype*. Consider the following example:

```
interface DS {
  interface IfElement {
    void      set (Object element);
    Object    get ();
  }
  interface IfContainer {
    void      insert(IfElement element);
    boolean   find(IfElement element);
  }

  class Element   implements IfElement;    // Syntactic extension
  class Container implements IfContainer;  // Syntactic extension
}
```

This example describes a simplified (partial) interface for a component encapsulating classes that provide basic data structure functionality. For a class to implement the `DS` interface (under our extension) it has to contain two publicly visible nested classes called `Element`, and `Container` with each of them conforming to the above interfaces (`IfElement`, `IfContainer`). For instance, consider a class `BinaryTree` that implements this interface:

```
class BinaryTree implements DS {
  public class Element implements DS.IfElement {
    public void set (Object element)    { ... } // implementations omitted
    public Object get ()                { ... }
  }

  public class Container implements DS.IfContainer {
    public void insert(DS.IfElement element)  { ... }
    public boolean find(DS.IfElement element) { ... }
  }
}
```

The "`implements DS`" clause in the class declaration makes the class conform to the `DS` interface. This entails the presence of two nested classes implementing the corresponding interfaces. It is worth noting that a class prototype may be declared to implement more than one interface and these interfaces do not have to be nested interfaces (as in our example). Additionally, there is no notion of inheritance among prototypes (i.e., a prototype declaration has no `extends` clause).

Note that the proposed scheme does not change the meaning of existing interface nesting (thus, no existing Java programs are affected by the changes). Also, no new keywords are required in the language.

It is interesting to ask whether our extension is sound. Informally, our rule to type interfaces containing class prototypes is very simple, and depends directly on the definition of interface conformance for regular (i.e., not extended) Java. Since a class cannot deviate at all from the exact signature of its declared interface, no room for unsoundness exists.

It is also worth mentioning a related typing extension, called *deep subtyping*, which was introduced by Wadler, Odersky, and this author in [21]. Simplifying slightly, a class `A` is a deep subtype of a class `B` if `A` is a subtype of `B` and for every nested class `F` of `B`, `A.F` is a deep subtype of `B.F`. Deep subtyping ensures soundness in the case of type variables ranging over a class and all its subtypes, and used to access nested classes. No such type variables can be defined in regular Java, but they can be used in parameterization extensions like GJ [4].

4 Advanced Applications of Interfaces for Nested Classes

Section 3 introduced a non-intrusive way for adding typing capabilities for nested classes into Java. Although the examples offered this far demonstrate the need for type checking nested classes, they are only the tip of the iceberg. In this section we argue that type checking nested classes is invaluable when combined with powerful programming constructs, like parameterization. Even though Java (with extensions) is used for illustration purposes, the argument is equally valid for other object-oriented languages.

4.1 Nested Classes and Constrained Parameterization

Java classes are second-class entities: they cannot be assigned to variables or passed as arguments to functions but there are language mechanisms that manipulate classes (most notably, inheritance). Type systems exhibit their benefits mainly in the presence of variability (for instance, arguments of functions are unknown but have a specific type). Hence, one would expect that interfaces will be useful in the case of *class functors*: functions with class arguments and/or producing new classes. Class functors are commonly implemented using parameterization (templates). Several parameterization mechanisms for Java have been proposed (e.g., [1], [4], [6], [11], [18]). Most of them allow expressing constraints on the parameters of a parameterized class. Common forms of constraints include `implements` and `extends` clauses, specifying an interface that the actual parameter must implement or a superclass that it must inherit from. `implements` clauses can be used in conjunction with our proposed type system extension, to enable the compiler to statically type check more programs, by offering type guarantees for nested classes.

Consider, for instance, the `DS` interface of Section 3.2. We could use its definition in a constraint for a parameterized class. (The syntax used in this example is that of GJ [4].)

```
public class Dictionary<DataStructure implements DS> {
  DataStructure.Container dict;
  public void insert(Object o) {
    DataStructure.Element e = new DataStructure.Element();
    e.set(o);
    dict.insert(e);
  }
  ...
}
```

The `Dictionary` class implements an adaptor that forces a specific (general) interface on any data structure that implements `DS`. Our extensions allow the compiler to statically type check the above definition, without any knowledge of the specifics of the actual parameter of `Dictionary`. The `DS` interface offers enough type information to ensure that nested classes `DataStructure.Container` and `DataStructure.Element` are defined, and that all method calls (e.g., `dict.insert(e)`) are legal.

4.2 Mixins and Nested Classes

In the previous section, we saw how interfaces for nested classes can be used in conjunction with parameterization. Here we will examine an interesting special case with examples taken from the recent research literature.

A very useful kind of class functors in object-oriented languages comes in the form of mixin classes (or "mixins"). Mixins [3] can be viewed as classes whose superclasses are left unspecified at mixin definition time.[2] The superclass is later specified during mixin *composition*. This property of mixins allows them to encode incremental extensions to several different class hierarchies. It is easy to see from the above description that a mixin can be modeled as a class functor: a function that takes a class argument (the superclass) and returns another class (the corresponding subclass). This is, in fact, one of the methods used for actual implementations of mixins in the context of languages supporting classes as first-class entities (e.g., using the reflective capabilities of Smalltalk).

Mixins can be simulated using parameterization. For instance, in C++, a mixin class can be a class parameterized with respect to its superclass as shown below:

```
template <class Super> class Mixin : public  Super { ... /* Mixin members */ };
```

Unfortunately, there is no way of constraining a parameterization in C++. Hence, such mixins, when viewed as functions on classes, are inherently untyped.

The Java language does not support mixins but this has been a topic of interest for researchers [1][6]. The parameterization mechanism of Agesen et.al. [1] is superficially similar to C++ templates but allows parameterizations to be constrained using interface specifications. The work of Flatt, et al [6] emphasizes the design, rather than the implementation, of a mixin facility for Java. Mixin arguments in [6] are explicitly constrained by specifying an *inheritance interface* for each mixin. Here we will use the syntax of [1] as it is more complete for our purposes. As an example, consider the following interface and mixin definitions:

```
interface Foo2 {
  Foo2     meth1 ();
}
class Mix <Super implements Foo2> extends Super {
  Foo2 get_foo() { return super.meth1(); }
  ...
}
```

The `implements` clause in the mixin definition specifies that the mixin parameter (i.e., the superclass of the produced class) should conform to interface `Foo2`. The need for conformance is evident in the body of method `get_foo`: the code calls a method `meth1` in the mixin's superclass.[3]

Mixins are of much more than purely academic interest. The work of VanHilst and Notkin [19][20] demonstrated the soft-

2. There have been other uses of the term "mixin" to describe specific language idioms that approximate this general mechanism. Most notably, CLOS classes whose superclass resolution relies on linearization of multiple inheritance and classes in a specific C++ multiple inheritance pattern (having a common virtual base class) are both called "mixins".

ware engineering aspects of using mixins in general object-oriented implementations. Their approach offers significant benefits (compared to *application frameworks* [8]) in the flexibility, efficiency, and composability of implementation entities. Although VanHilst and Notkin did not realize that these benefits were afforded by the mixin-like character of their technique, this became evident later with the introduction of the alternative approach of *mixin layers* [14][15][16]. Mixin layers consist of nested mixin classes such that the parameter of the outer mixin determines the parameters of all inner mixins. For instance, consider the following mixin layer definition:

```
class LayerThis <LayerSuper> extends LayerSuper {
  public class First  extends LayerSuper.First  { ... }
  public class Second extends LayerSuper.Second { ... }
  public class Third  extends LayerSuper.Third  { ... }
}
```

As can be seen, a mixin layer is a mixin which contains nested classes that inherit from classes nested inside the class that parameterizes the layer. As shown in [14] and [15], the mixin layers approach solves the scalability problems of the VanHilst and Notkin technique. Mixin layers represent an elegant way to implement object-oriented programs from reusable components.

Expressing mixins with parameterization enables us to use techniques similar to those of Section 4.1. Interfaces for nested classes are ideal for use in conjunction with mixin layers. An example from the domain of data structures was used in [15] to demonstrate the mixin layers approach. Four data structure layers were defined, containing refinements for the `Container` and `Element` classes. Here we would like to define generalized interfaces for allocators and data structures (e.g., binary trees, hash tables, lists). Interface conformance will serve as a static check of the interchangeability of the corresponding mixin layers. This could be effected with the following interface declarations (note that the first is reproduced from Section 3):

```
interface DS {
  interface IfElement {
    void      set (Object element);
    Object    get ();
  }
  interface IfContainer {
    void      insert(IfElement element);
    boolean   find(IfElement element);
  }

  class Element   implements IfElement;
  class Container implements IfContainer;
}


interface ALLOC {
  interface IfElement { }
  interface IfContainer {
    IfElement    alloc_node();
  }
  class Element   implements IfElement;
  class Container implements IfContainer;
}
```

Consider now an example mixin layer defining a binary tree. We would like to constrain the mixin parameter so that its valid values are classes supporting the `ALLOC` interface.

---

3. In this case, the dependency could be inferred from the code. That is, by analogy to many other forms of polymorphism in programming languages, the mixin could be considered a polymorphic entity that can be parameterized by any class specifying a method `meth1` with a compatible type signature. We will discuss polymorphism and type inference in more detail in Section 5.

```
class BinaryTree <Alloc implements ALLOC> implements DS extends Alloc {
  class Element implements DS.IfElement extends Alloc.Element {
    public void set (Object element)    { ... } // Implementation omitted
    public Object get ()                { ... }
  }

  class Container implements DS.IfContainer extends Alloc.Container {
    public void insert(DS.IfElement element)  { ... }
    public boolean find(DS.IfElement element) { ... }
  }
}                                                                                    (1)
```

The constraint on the mixin parameter ("`Alloc implements ALLOC`") prevents the `BinaryTree` layer from being instantiated with classes that will result in invalid compositions.

4.3 Virtual Types and Nested Classes

Virtual types are a programming language mechanism that enables the propagation of type information from a subclass to a superclass. More specifically, when a subclass redefines a virtual type (originally declared by its superclass) the redefined version is used for type-checking all superclass code that utilizes the virtual type. The scope of a virtual type is (usually) delimited by the class that defines it. Hence, class nesting plays a natural role in the use of virtual types (e.g., see the examples in [9]).

Virtual types were introduced as virtual patterns in the BETA programming language [10] but they have since been used in various contexts (e.g., [5], [18]). There have been several attempts at unifying virtual types and parametric types, two of which [5][21] use class nesting extensively. Here we will only discuss the proposal for statically safe virtual types by Bruce, Odersky, and Wadler [5], because the code directly supports our argument in favor of interfaces for nested classes. In that work, class nesting is used to group mutually referential classes together, so as to delimit the scope of their definitions.

Consider the following example from [5]. This defines an interface for a list collection class, whose elements are either characters or floating point numbers, in alternating order. The form of virtual types described in [5] ensures that the definition can be type-checked statically even though refinements (i.e., subinterfaces) of the general list interface shown can give new meaning to the `XThis` and `YThis` types.

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

The syntax of this example is an extension of Java: parenthesized identifiers (following interface names) designate an interface name with "virtual type" properties—it changes automatically in extensions of the interface. The prefix `@` indicates that the type following it is *exact*: values must be of exactly this type and not any subtype of it. The reader should consult [5] for more details.

Our proposed interfaces for nested classes are useful in cases like the above. Bruce, Odersky, and Wadler explicitly state in [5]: "[A type variable that extends `AltListGrpIfc`] must support inner interfaces with variable names `XThis` and `YThis`." Unfortunately, this is not the case in Java. As we have already seen, interface nesting has no semantic consequences on classes that implement the interface. Instead, nesting only has namespace significance (i.e., it delimits the scope of inner interface declarations). Our proposal of Section 3 introduces explicit syntax so that both requirements are accommodated and existing Java programs are not affected. There are many ways in which our extension may interact with the

syntactic additions of [5]. One way would be to imitate the class declarations of [5]. Then the above example becomes:

```
public interface AltListGrpIfc {
  public interface XListIfc (XThis) {[same as before] }
  public interface YListIfc (YThis) {[same as before] }

  class XList(XThis) implements XListIfc;
  class YList(YThis) implements YListIfc;
}
```

5 Related Work

A class containing nested classes can be considered a large scale component. Extending the interface functionality to include such classes is a significant step in increasing the granularity of reusable software entities. Type signatures for multiple-class components have been studied before. The GenVoca model of software construction [2] defines the idea of a component's *realm*. A realm identifies the set of all components that are interchangeable in a parameterization, exactly like our extended interfaces. In fact, the concept of a realm is reified as a programming language construct in the P++ language [13].

As we mentioned before, interfaces can be viewed as explicit types for classes. From a programming language standpoint it makes sense to ask whether the type of a class can be inferred from its definition. This is (to an extent) true in all the examples we discussed. Consider, for instance, the code fragment (1) from Section 4.2. Both requirements on the mixin parameter can be inferred from the mixin layer definition:

- The `Element` nested class has a superclass called `Element`, nested inside the mixin parameter (`Alloc.Element`)
- The `Container` nested class has a superclass called `Container`, nested inside the mixin parameter (`Alloc.Container`). The additional requirement that this superclass provide a method `alloc_node` is expected to be deducible from the definition of the `insert` method (omitted in (1)).

In other words, instead of using explicit constraints on classes, we could consider them polymorphic: they assume the most general type permitted by their definitions. Nested classes are no different than other members of the class when it comes to type inference. Type inference for record members was examined, for instance, in [12]. Combining this technique with mixin typing (e.g., [6]) should result in a type inference technique that can handle the typing requirements of nested classes and mixins.

6 Conclusions

We discussed type system additions to class-based object oriented languages for dealing with nested classes. The concrete application of our work is an extension to the interface mechanisms of Java. The extension does not change the semantics of existing Java programs and adds no new keywords to the language. Additionally, we argue that type system support for class nesting is particularly useful when classes are used in complex contexts, such as those explored in the recent research literature. We expect that these ideas will become increasingly important both in the context of Java as well as in the design of future object-oriented languages.

References

[1] O. Agesen, S. Freund, and J. Mitchell, "Adding Type Parameterization to the Java Language", *OOPSLA 1997*, 49-65.

[2] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodologies*, October 1992.

[3] G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

[4] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, "Making the future safe for the past: Adding Genericity to the Java Programming Language", *OOPSLA 1998.*

[5] K.B. Bruce, M. Odersky, and P. Wadler, "A Statically Safe Alternative to Virtual Types", *ECOOP 1998*. Earlier version by K.B. Bruce and M. Odersky, *Workshop on the Foundations of Object-Oriented Languages (FOOL 98).*

[6] M. Flatt, S. Krishnamurthi, M. Felleisen, "Classes and Mixins", ACM *Symposium on Principles of Programming Languages (PoPL)*, 1998.

[7] JavaSoft, "Inner Classes Specification", from `http://java.sun.com/products/jdk/1.1/docs/`.

[8] R. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, 1(2): June/July 1988, 22-35.

[9] O.L. Madsen and B. Møller-Pedersen, "Virtual classes: A powerful mechanism in object-oriented programming", *OOPSLA 1989*, 397-406.

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

[11] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice", *ACM Symposium on Principles of Programming Languages (PoPL)*, 1997.

[12] A. Ohori, "A Compilation Method for ML-style Polymorphic Record Calculi", *ACM Symposium on Principles of Programming Languages (PoPL)*, 1992.

[13] V. Singhal, "A Programming Language for Writing Domain-Specific Software System Generators", Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, August 1996.

[14] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components", *Int. Conf. on Softw. Reuse (ICSR)*, 1998.

[15] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998.*

[16] Y. Smaragdakis, *Implementing Large-Scale Object-Oriented Components*, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, Dec. 1999.

[17] B. Stroustrup, *The C++ Programming Language, 3rd Edition*, Addison-Wesley, 1997.

[18] K. Thorup, "Genericity in Java with Virtual Types", *ECOOP 97*, 444-471.

[19] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[20] M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996.*

[21] P. Wadler, M. Odersky and Y. Smaragdakis, "Do Parametric Types Beat Virtual Types?", unpublished manuscript, posted in the Java Genericity mailing list (java-genericity@cs.rice.edu), October 1998.