

# Static Virtual Types in Java Layers

Richard Cardone Calvin Lin  
Department of Computer Sciences  
University of Texas at Austin  
{richcar, lin} @cs.utexas.edu

December 20, 2000

## Abstract

This specification describes how a simple, static form of virtual typing can be implemented in a parametrically polymorphic version of Java. We discuss the need for virtual types and the tradeoffs of integrating virtual types into a statically typed, object-oriented language. Though our discussion is couched in terms of Java Layers, the design can be used in more general settings.

## 1 Introduction

This document describes how a restricted form of virtual typing can be integrated into a parametrically polymorphic implementation of Java. To understand the usefulness and flexibility of virtual typing, consider *recursive classes*, which are classes that use their own type in their definition. When a recursive class is subclassed, we often want some or all of the uses of the recursive class's type to be replaced by uses of the subclass's type. This is what happens in the following example, which contains two recursive classes:

```
class Node          class Db1Node extends Node
{
  Node next;
}
{
  Db1Node prev;
}
```

In the code above, the `next` field in `Db1Node` is of type `Node`, while its `prev` field is of type `Db1Node`. In subclassing `Node`, the programmer probably wants a class in which all node fields are typed as `Db1Node`, but this is difficult to achieve using standard inheritance. One way to solve this problem is to use virtual types, which automatically replaces a general type with one of its subtypes. In the example above, the `next` field in `Node` would be virtually typed. When `Db1Node` inherits from `Node`, the `next` field would automatically have its type transformed to `Db1Node`. *Virtual typing* is the automatic adaptation of types through inheritance.

Virtual types enhance the precision of an object-oriented language's type system by allowing specialized types to automatically replace more general types based on context. This precision leads to better type checking and less manual typecasting. For example, with virtual typing, the `next` field of the `Db1Node` class would not have to be typecast to get its precise type; it would already have type `Db1Node`. Also, the compiler could guarantee through type checking that `next` was always assigned a `Db1Node` type.

### 1.1 A Specialized Form of Virtual Typing

A specialized form of virtual typing allows the type of Java's *this* reference to be used in class definitions. In its dynamic form, the type of *this* resolves to the class type of the object constructed at runtime. The

type of *this* is virtual because subclasses—the types that are actually constructed at runtime—are unknown at the time that a class is defined.

The ability to express the type of *this* is especially desirable in Java Layers (JL) because deep class hierarchies create many opportunities for its use. Under the JL programming methodology, application features are encapsulated in individual classes that are then composed to build complete applications. Class composition often involves the use of inheritance, which tends to create applications with deep class hierarchies. To maximize reuse potential, classes are designed to make as few assumptions as possible about the compositions in which they will participate. By implementing the virtual type of *this*, JL allows programmers to refer to the product of any composition from within the classes that make up the composition. This enhanced expressiveness is achieved without reducing the ability to compose classes.

JL supports a static implementation of the virtual type of *this*. By static, we mean that the type of *this* is determined when polymorphic types are instantiated rather than when runtime objects are constructed. This approach promotes static type checking since types are known before runtime. On the other hand, the static approach only approximates the type of *this* in cases where an object's exact type is not decidable until runtime. The limitations of JL's static implementation are described in Section 6.1, Static Binding.

JL's contribution is to show that statically evaluated virtual types are both useful and easy to implement. In object-oriented languages that support parametric polymorphism, virtual types can be evaluated at instantiation time, which requires no modification to the language's type system.

## 1.2 Java Layers Background

In its first implementation, Java Layers [8] supported constrained parametric polymorphism only in the *layer* construct. The *thisClass* construct was used to provide virtual typing inside layer definitions. In the second and current version of Java Layers [9], which we refer to simply as Java Layers in this specification, the specialized layer construct has been eliminated and type parameters are supported in class and interface definitions. This more general approach to parametric polymorphism necessitates a more general approach to virtual typing. The design of this general virtual typing mechanism, which we call static virtual typing, is the subject of this specification.

This discussion proceeds as follows. We talk about Related Work in Section 2. We introduce some examples in Section 3 before describing our virtual type proposal for parameterized classes in Section 4. Section 5 discusses how virtual types work with nested classes and interfaces. Section 6 discusses implementation issues and Section 7 contains our conclusions.

## 2 Related Work

Virtual types have been proposed for Java by Thorup [14] as a way to provide genericity. Thorup's general implementation of virtual types has more than enough expressive power to solve the problem described above in the Node example. Thorup's proposal also allows the type of *this* to be expressed.

A more common approach to adding generics to Java, however, is to implement some form of constrained parametric polymorphism [7][13]. Unfortunately, adding type parameters to Java does not allow type adaptation through inheritance—types declared in parent classes are fixed and do not automatically change through inheritance. Adding type parameters, therefore, does not address the problem described in the Node example.

Bruce and colleagues [5][6] have proposed a way to augment a parametrically polymorphic implementation of Java with some of the capabilities of virtual typing. Their proposal introduces the *ThisType* construct and an exact type operator to augment Generic Java [4], a parameterized polymorphic implementation of Java. Support for *ThisType*, however, breaks the equivalence between subtyping and subclassing in Java and introduces the notion of *matching* for type checking. These changes give rise to a new programming model in which subclasses match their superclasses, but subclasses do not always subtype their superclasses.

In Java Layers, the need for mixins dictates the use of parametric polymorphism as the main vehicle for achieving genericity. On the other hand, as we described in the last section, JL would benefit from virtual typing inside layer compositions so that references to yet-to-be-defined subtypes could be expressed. JL satisfies this need for virtual typing by borrowing from both Thorup's virtual types and Bruce's *ThisType*. JL's limited virtual type mechanism combines the power of mixins, which are not present in Thorup's proposal, with a way to reference and use the type of *this* in some contexts. In contrast to Bruce's approach, JL does this without changing the semantic relationship between subclassing and subtyping in Java. The static approach described in this specification can be used generally to enhance parametrically polymorphic implementations of Java or similar object-oriented languages.

Czarnecki and Eisenecker [10] find a way to effectively reference the type of *this* in C++ template classes. Their approach involves the use of a separate, manually configured repository class that is passed in as a type parameter during template instantiation. The type of the instantiation itself is specified in the configuration class. As in JL, all processing takes place statically. The type of *this* can be referenced from within the template instantiation, but the type's members cannot all be accessed due to the way C++ processes templates.

A form of virtual typing has also been implemented in domain-specific GenVoca [2] generators. In these domain-specific generators, the compiler usually has built-in knowledge about which type specifications are virtual. In some GenVoca implementations, such as P3 [3], a programmer visible mechanism is also available. We now briefly describe virtual types in Java Layers implementations, a domain-independent outgrowth of GenVoca research.

### 3 Examples

We now illustrate JL's virtual typing mechanism by looking at a number of examples. We describe how virtual types are expressed and how they can be used.

#### 3.1 Simple Instantiations

The `Node1<>` class in Figure 1 uses the *This* virtual type. *This* can only be used in parameterized class or parameterized interface definitions. Intuitively, *This* can be thought of as the class type of the *this* reference. The precise meaning of *This*, however, is slightly more complex since *This* can appear in both classes and interfaces and, as mentioned in the Introduction, *This* is implemented statically. In this section, we discuss how *This* is bound when it appears in a class and defer a more general discussion of *This* binding semantics until Section 4.1.

```
// Base parameterized class.
class Node1<>
{
    This next;
    This getNext(){..}
    void setNext(This n){..}
}

// Instantiation of Node1<> in some class.
class SomeClass{... Node1<> node; ...}
```

**Figure 1 - Node1 Definition and Use**

*This* is always bound when a parametric type is instantiated. Instantiations of parametric classes create class hierarchies that either have parametric or non-parametric classes at their leaf nodes. By default, these leaf nodes are the types bound to *This* in all parametric classes in the hierarchy. For example, in Figure 1, `Node1<>` is instantiated in the body of `SomeClass` by specifying an empty parameter list (since `Node1<>` does not define any explicit type parameters). The class hierarchy generated by this instantiation consists only of `Node1<>` itself, so `Node1<>` is the leaf node whose type is bound to all occurrences of *This* in the hierarchy.

On the other hand, `DNode1Fixed` shown in Figure 2 is a non-parameterized class that extends `Node1<>`. When a non-parametric class inherits directly from a parametric class, the non-parametric class becomes the leaf node in the hierarchy generated by the instantiation. In this example, the hierarchy consists of two classes, `DNode1Fixed` and its parent `Node1<>`. All occurrences of *This* in `Node1<>` are replaced by `DNode1Fixed`.

```
// Non-parameterized double linked list node.
class DNode1Fixed extends Node1<>
{
  DNode1Fixed prev;
  DNode1Fixed getPrev(){..}
  void setPrev(DNode1Fixed n){..}
}
```

**Figure 2 - A Non-Parameterized Node Class**

### 3.1.1 Examples of Deeper Hierarchies

Figure 3 defines `DNode1<>`, a parameterized class that implements a double linked list node by subclassing `Node1<>`. The virtual type *This* in both classes are replaced by the same type whenever `DNode1<>` is instantiated. `DNode1FixedDeep`, a non-parameterized class defined below to extend `DNode1<>`, represents such an instantiation. Here, all occurrences of *This* in `Node1<>` and `DNode1<>` will be replaced by `DNode1FixedDeep` in this instantiation.

```
// Parameterized double linked node.
class DNode1<> extends Node1<>
{
  This prev;
  This getPrev(){..}
  void setPrev(This n){..}
}

// Non-parameterized double linked node.
class DNode1FixedDeep extends DNode1<>
{}
```

**Figure 3 - A Parameterized Node Class**

## 3.2 Instantiations with Constrained Type Parameters

The example in Figure 4 illustrates the syntax and binding semantics for constrained type parameters in JL.

```
// Parameterized parent class
class Node2<T implements Ifc>
{
  T data;
  This next;
  This getNext(){..}
  void setNext(This n){..}
}

// Parameterized child class.
class DNode2<T implements Ifc> extends Node2<T>
{
  This prev;
  This getPrev(){..}
  void setPrev(This n){..}
}
```

**Figure 4 - Classes with Explicit Type Parameters**

Node2<T> and DNode2<T> are parameterized classes each of which take a type parameter constrained by interface Ifc. Both classes use *This* in their class bodies. When DNode2<T> is instantiated, all occurrences of *This* in both classes will be bound to the same type. For example, the expression DNode2<MyData> instantiates a class by replacing T with MyData and by replacing *This* with DNode2<MyData> everywhere in the definitions of Node2<T> and DNode2<T>. The constraint on the type parameter T in both parametric classes requires that MyData be a class that implements interface Ifc.

## 4 Types

In JL, a parameterized type can be a parameterized class or a parameterized interface. Parameterized types are not types in JL, but instead are considered *type functions*. *Instantiations* of parametric types are expressions in which all required actual type parameters are supplied so that all formal type parameters can be bound. Instantiations of parametric types are Java types.

The virtual type *This* is an implicit type parameter associated with all parametric types. *This* is not defined and cannot be used in non-parametric type definitions. *This* can appear, with few restrictions, wherever a class type is valid inside the body of a parameterized type definition. The binding semantics of *This* give its virtual typing power.

### 4.1 How *This* is Bound

*This* is bound implicitly or explicitly whenever a parametric type is instantiated. The class type bound to *This* is called the *This-binding*.<sup>1</sup> A programmer may explicitly designate a *This-binding* by instantiating a parameterized type with an additional actual parameter. More precisely, whenever a parameterized type defined with *n* formal type parameters is instantiated with *n + 1* actual type parameters, the first actual parameter is bound to *This* if it's a class type. *This* is never bound to an interface type; any attempt to do so results in an error.

We now look at examples of explicit *This*-bindings. Consider the following field definitions which refer to the Node1 example from Section 3.1:

```
Node1<DNode1Fixed> fld1;  
Node1<Node1<>>      fld2;  
Node1<>             fld3;
```

**Figure 5 - Instantiated Types**

The fld1 instantiation explicitly binds class DNode1Fixed to *This*, which has the effect replacing all occurrences of *This* within the body of Node1<> with DNode1Fixed. The fld2 instantiation binds class Node1<> to *This* explicitly, which is what happens by default in the fld3 instantiation.

If a binding is not explicitly designated, *This* will be bound implicitly. The default binding depends on the context of the instantiated type and on whether the instantiated type is a class or interface. Instantiated types may appear in a **standalone**, **supertype** or **constraint** context. We now describe how *This* is bound in each context.

#### 4.1.1 Standalone Context

Instantiations are in a standalone context when they occur outside any extends or implements clause. This includes instantiations in field or variable declarations, as formal parameters or return types in methods, in cast, instanceof or allocation expressions, or as actual type parameters in other type instantiations. In a standalone context, the implicit *This-binding* for a parameterized class is the class itself and the implicit *This-binding* for a parameterized interface is java.lang.Object.

---

<sup>1</sup> See the Appendix for a short discussion on enhancements to *This-binding* semantics under consideration.

The `fld3` instantiation from Figure 5 illustrates an implicit `This`-binding for the parameterized class `Node1<>` in which `This` is bound to the type being instantiated. The resulting type is represented canonically as `Node1<Node1<>>`, where the `This`-binding is listed as the first type parameter (just as if it were explicitly specified). To avoid infinite recursion in the notation, the `This`-binding type does not specify its own `This`-binding when it refers to the type being instantiated.

We now look at an example of `This`-binding in a parametric interface. Figure 6 shows the definition of interface `Ifc<>` and its use in an `instanceof` expression in method `m2()`. The `instanceof` expression instantiates `Ifc<>` and implicitly binds `java.lang.Object` to `This`. The actual type instantiated for the `instanceof` operation is `Ifc<Object, HashMap>`.

```
// Parameterized interface with a single method
// declaration.
interface Ifc<T implements SortedMap>
{void m1(This, T);}

// Method in some class.
void m2(){... if (x instanceof Ifc<HashMap>)...}
```

**Figure 6 - Parametric Interface**

### 4.1.2 Supertype Context

Instantiations are in a supertype context when they occur (1) in an `extends` or `implements` clause of a class definition or (2) in an `extends` clause of an interface definition. In a supertype context, the type being defined (the *defining subtype*) inherits from the instantiated type (the *instantiated supertype*). If the defining subtype is a non-parameterized class, then `This` in the instantiated supertype is bound to the defining subtype itself. If the defining subtype is a parameterized class, then `This` in the instantiated supertype is bound to the `This`-binding of the defining subtype.

When the defining subtype is an interface, the instantiated supertype must also be an interface. If the defining subtype is a non-parameterized interface, then `This` in the instantiated supertype is bound to `java.lang.Object`. If the defining subtype is parameterized interface, then `This` in the instantiated supertype is bound to the `This`-binding of the defining subtype. Table 1 summarizes implicit binding in the supertype context.

Defining Subtype	This-Binding in Instantiated Supertype
Non-Parametric Class	Class being defined.
Parametric Class	This-binding of class being defined.
Non-Parametric Interface	<code>java.lang.Object</code> .
Parametric Interface	This-binding of interface being defined.

**Table 1 - Implicit Bindings in Supertype Context**

An example of each of the four cases is given below. In Figure 7, non-parametric class `D` inherits from type `C<D>`. `This` is bound to `D` in the body of `C<>`.

```
class C<>{...}
class D extends C<> {...}
```

**Figure 7 - Non-Parametric Class**

In Figure 8, the declaration of field `e` in some class would instantiate types `E<E<>>` and `C<E<>>`. *This* is bound to `E<>` in the bodies of `C<>` and `E<>`.

```
class C<> {...}
class E<> extends C<> {...}
E<> e;
```

**Figure 8 - Parametric Class**

In Figure 9, non-parametric interface `J` inherits from type `I<Object>`. *This* is bound to `Object` in the body of `I<>`.

```
interface I<> {...}
interface J extends I<> {...}
```

**Figure 9 - Non-Parametric Interface**

In Figure 10, the declaration of field `k1` in some class would instantiate types `K<Object>` and `I<Object>`. *This* is bound to `Object` in the bodies of both `I<>` and `K<>`. The declaration of field `k2` would instantiate types `K<String>` and `I<String>`. *This* is explicitly bound to `String` in the body of `K<>` and implicitly bound to `String` in the body of `I<>`.

```
interface I<> {...}
interface K<> extends I<> {...}
K<> k1;
K<String> k2;
```

**Figure 10 - Parametric Interface**

### 4.1.3 Constraint Context

Instantiations are in a constraint context when they occur in a constraint clause of a formal type parameter. Constraint clauses use the `extends` and `implements` keywords to restrict the actual parameters allowed in an instantiation. The `extends` and `implements` keywords are overloaded for use as type-based filters.

In constraint contexts, the types specified as constraints on formal type parameters are only used to restrict the type of actual parameters. Meaningful *This*-bindings cannot always be implicitly determined, so JL simply ignores the *This*-bindings of parametric types for the purposes of constraint filtering. To understand the rationale for this lenient approach, consider parametric class `F` defined below with two constrained type parameters. Actual parameter `T` is required to be a class that implements parametric interface `Ifc<>` and actual parameter `U` is required to extend parametric type `Typ<>`.

```
class F<T implements Ifc<>, U extends Typ<>> extends T {...}
```

Since actual parameter `U` can be a type completely unrelated to class `F`, it's not reasonable to assume that the *This*-binding for `F` would be appropriate as the *This*-binding for `U` or its supertype, `Typ<>`. The only cases in which default bindings might be reasonably assigned are those that involve mixins, such as for parameter `T` in class `F` above. For simplicity and with negligible loss of filtering power, JL uniformly ignores *This*-bindings during constraint checking.

### 4.1.4 Binding *This* in Mixins

Since mixins play such an important role in Java Layers, we conclude this section with two binding examples using mixins. In Figure 11, the declaration of field `m1` in some class would bind *This* to `M<C<>>` in the bodies of both `M<T>` and `C<>`. Let `$A` stand for the string `M<C<>>`. The two types instantiated by declaring field `m1` are `M<$A, C<$A>>` and `C<$A>`.

```

class C<> {...}
class M<T> extends T {...}

M<C<>> m1;
M<M<M< C<> >>> m2;

```

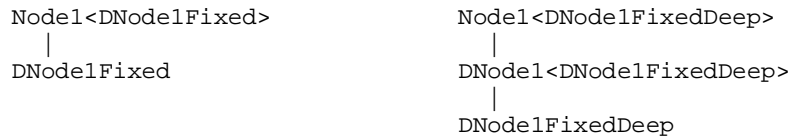
**Figure 11 - Mixins**

Figure 11 also shows the declaration of field `m2`. In this case, *This* is bound to `M<M<M< C<> >>>` in the bodies of both `M<T>` and `C<>`. Let `$B` abbreviate the string `M<M<M< C<> >>>`. The declaration of field `m2` instantiates four types:

1. `M<$B, M<$B, M<$B, C<$B>>>>`
2. `M<$B, M<$B, C<$B>>>`
3. `M<$B, C<$B>>`
4. `C<$B>`

## 4.2 Subtypes

Referring to examples from Section 3.1 on page 3, the following class hierarchies are formed by `DNode1Fixed` and `DNode1FixedDeep`:



There is no type relation between `DNode1Fixed` and `DNode1FixedDeep`, other than they both descend from `Object` (not shown). Parameterized types are not complete types, so they don't belong to any type hierarchy—only instantiations belong to type hierarchies. Parameterized types, however, do form their own hierarchies of generics or type functions that do obey the standard Java rules of inheritance. In JL, these hierarchies are not integrated into the type system so they play no role at runtime.

## 5 Nested Parameterized Types

All parameterized types have their own `This`-binding; this is true even if one parameterized type happens to be nested within another. Consider a typical Java Layers example using mixin layers:

```

// A nested class
class Collection<D>
{
    static class Node<D> {...}

    static class Container<D>{Node<D> n1;}
}

// A mixin
class Mix<D, C extends Collection<D>>
extends C
{
    static class Node<D> extends C.Node {...}

    static class Container<D> extends C.Container
    {Node<D> n2;}
}

// Instantiation
Mix<Data, Collection<Data>> m;

```

**Figure 12 - Nested Types Without Virtual Typing**

The `Collection<D>` and `Mix<D>` classes each contain two parameterized classes, `Node<D>` and `Container<D>`. The declaration of field `m` in some class instantiates both enclosing classes, `Collection<D>` and `Mix<D>`, and all their nested classes. Using rules of implicit binding, Table 2 lists the `This`-bindings for each class instantiated by the declaration of field `m`. `$A` stands for `Mix<Data, Collection<Data>>`.

Class	This-Binding
<code>Collection&lt;Data&gt;</code>	<code>\$A</code>
<code>Collection&lt;Data&gt;.Node&lt;Data&gt;</code>	<code>\$A.Node&lt;Data&gt;</code>
<code>Collection&lt;Data&gt;.Container&lt;Data&gt;</code>	<code>\$A.Container&lt;Data&gt;</code>
<code>Mix&lt;Data, Collection&lt;Data&gt;&gt;</code>	<code>\$A</code>
<code>Mix&lt;Data, Collection&lt;Data&gt;&gt;.Node&lt;Data&gt;</code>	<code>\$A.Node&lt;Data&gt;</code>
<code>Mix&lt;Data, Collection&lt;Data&gt;&gt;.Container&lt;Data&gt;</code>	<code>\$A.Container&lt;Data&gt;</code>

**Table 2 - This-Bindings for Mixin Layers**

Note that fields `n1` and `n2` are not virtually typed; `n1` will always be of type `Collection<D>.Node<D>` and `n2` will always be of type `Mix<D,C>.Node<D>`. If the intent is to use the most derived subclass of `Node<D>` for both of these fields, we need to employ the *This* virtual type. Unfortunately, the simple use of `This.Node<D>` as the type for `n1` or `n2` result in references to nonexistent types. For example, in the case of `n1`, the type referenced by `This.Node<Data>` is `$A.Container<Data>.Node<Data>`, but `Container<Data>` does not have a nested `Node<Data>` subtype.

A solution to this problem of resolving *This* references in nested types is shown below in Figure 13. The *This* keyword is qualified with the type name of a lexically enclosing type, `Collection`. The qualified virtual type, `Collection.This`, refers to the `This`-binding of `Collection<D>` when used inside `Collection<D>.Container<D>`. The qualified virtual type, `Mix.This`, refers to the `This`-binding of `Mix<D>` when used inside `Mix<D>.Container<D>`. In the instantiation of field `m`, both `Collection.This` and `Mix.This` refer to `$A` as defined in Table 2, which does have a `Node<Data>` nested class.

```
// A nested class
class Collection<D>
{
    static class Node<D> {...}

    static class Container<D>
    {Collection.This.Node<D> n1;}
}

// A mixin
class Mix<D, C extends Collection<D>>
extends C
{
    static class Node<D> extends C.Node {...}

    static class Container<D> extends C.Container
    {Mix.This.Node<D> n2;}
}
```

**Figure 13 - Nested Types With Virtual Typing**

An alternative solution to resolving *This* references in nested types involves revising our definition of *This*. Nested classes could be defined to have the same `This`-binding as their enclosing class. Using this approach, there would be one `This`-binding for a type and all its nested types. Fields `n1` and `n2` in Figure 12 would be declared with type `This.Node<D>` and the `This`-binding for all six classes listed in Table 2

would be \$A. JL1 took this approach in its implementation of *thisClass*. In deeply nested classes, this addressing approach resembles the specification of full path names in a hierarchical file system.

The first solution given above is more object-oriented and conforms to Java's method of qualifying *this* in inner classes. For these reasons, the first approach is chosen for JL.

## 6 Implementation

We now describe some of the issues pertinent to implementing *This* in JL.

### 6.1 Static Binding

Limiting *This* to use in parameterized types implies that a source-to-source translation from JL to standard Java is possible. This means that JL's virtual type support does not require changes to the Java type system and that virtual types can be statically type checked. On the other hand, it also means that *This* does not always precisely reflect the semantics of expressing the type of *this*. Recall, for example, that `DNode1Fixed` defined in Figure 2 is a non-parameterized class that extends `Node1<>`, a parameterized class. If class `DNode1FixedChild` were defined as a subclass of `DNode1Fixed`, all occurrences of *This* in `Node1<>` would still be bound to `DNode1Fixed`. Thus, the type implicitly bound to *This* at instantiation time is sometimes a superclass of the class of the actual runtime object.

### 6.2 Strategy

Java Layers can be built on top of most Java implementations that have already been extended with constrained parametric polymorphism. The extension proposed by Agesen et al. [1] supports mixins and, therefore, provides the best foundation for JL.

There are three implementation strategies under consideration for static virtual typing in JL.<sup>2</sup> The first strategy follows Agesen's [1] approach for implementing parameterized polymorphism in Java. A parameterized type `P<...>` is compiled into file `P.class` in an extended class file format. A specialized class loader is then used to interpret the extended format. This approach increases the work performed at load time, but potentially decreases memory usage and the total number of bytes read in at runtime.

Under this approach, a parametric type is instantiated *heterogeneously* which results in different compiled code for instantiations with different actual parameters. When an instantiation is encountered, the parameterized type's extended class file and all its argument types are loaded, recursively if necessary. If *This* is not explicitly bound, the rules for implicit *This*-binding are invoked to complete the binding of all type parameters. Type parameters are then checked against their constraints, which may require the loading of more types. If no constraints are violated, then an executable representation of the instantiation is created. The executable is in standard bytecode format.

A second implementation strategy uses an extended bytecode format for parameterized types as in the first strategy, but instantiations are handled at compile time rather than at load time. Parametric types are still instantiated heterogeneously, but all such processing takes place as instantiations are encountered during compilation. Under this approach, the compiler needs to understand the extended bytecode format, but the runtime loader doesn't need to because it processes only fully instantiated types. Thus, parametric polymorphism and the *This* virtual type can be implemented without any change to the runtime environment. This approach also gives the JL compiler the opportunity to apply class hierarchy optimization to instantiated types. On the other hand, compile time instantiation can lead to increased runtime memory usage if multiple copies of nearly identical code are loaded at the same time.

The third implementation strategy under consideration represents an amalgamation of the first two strategies. Under this approach, JL would support both load time and compile time instantiation. At runtime, class loaders would first look for pre-compiled instantiations and, only if this fails, would runtime

---

<sup>2</sup> A fourth strategy, which uses source-to-source translation only, may be useful for prototyping.

instantiation be attempted. This strategy allows the programmer to decide when types should be instantiated, but requires more work to implement.

### 6.3 Names

Instantiations must be given unambiguous, legal type names since they are standard Java types. These type names (and their associated class file names) are comprised of three components:

- the parametric type name.
- the name of the actual types associated with each formal type parameter.
- the name of the This-binding type.

An instantiation's type name must be appropriately mangled to conform to Java naming conventions, though we will continue to use our type naming notation for expositional purposes. A simpleminded implementation of a type naming scheme would lead to a proliferation of identical types under different names. Consider, for example, the declaration below of fields `v1` and `v2` in some class.

```
interface Ifc<T> {T get();}

Ifc<String>          v1;
Ifc<Integer, String> v2;
```

The types generated by `v1` and `v2` are `Ifc<Object, String>` and `Ifc<Integer, String>`, with the `Object` and `Integer` types representing `This`-bindings. Since `Ifc<T>` doesn't reference `This` in its body, nor does any of its supertypes, the two generated types are equivalent in every way except for the `This`-binding type, which is never used. An implementation of parametric polymorphism without the virtual type `This` would use the same type name—`Ifc<String>`—for both instantiations of `Ifc<T>`. Since all type checking in Java is based on names, naming is not only a matter of efficiency, but also a matter of program semantics: We would like to be able to treat `v1` and `v2` as fields of the same type. This only can be accomplished if the type names for both are the same.

JL solves this problem by not including the `This`-binding type in names of instantiated types that don't use `This` in their bodies or in the bodies of any of their supertypes. In these cases, the naming reverts back to that of a parametrically polymorphic Java without virtual typing. Whenever an instantiated type needs to be loaded, the loader first checks for the type without a `This`-binding component in its name. If that fails, a load is attempted for the type name with the appropriate `This`-binding component included. A parametric type is either dependent or not dependent on `This`, so only one of the names will ever exist at a time. This dependency information is part of the extended class file format generated by the compiler for parametric types.

### 6.4 Usage Restrictions

Instantiated types can be used freely in almost all circumstances that non-parametric types can be used. Support for local and anonymous nested classes, however, requires special consideration. Explicit `This` bindings must be used whenever instantiations are used as supertypes of local or anonymous classes, and the bindings specified in these circumstances cannot be classes with local scope. Also, anonymous classes cannot be parameterized due to their spare syntax; local classes may be parameterized, but the use of `This` in their bodies is prohibited. Instantiations that appear in constraint clauses of parameterized local classes must be specified with explicit `This`-bindings.<sup>3</sup>

---

<sup>3</sup> Going out on a limb here, we think that Java's local and anonymous classes were a bad idea because of the complexity they introduced to the language, so we probably won't enhance them at all when implementing JL.

## 6.5 Reflection

Soloranzo and Alagic [11] argue the importance of faithfully providing accurate reflective information for parameterized types and their instantiations. Our proposed implementation of JL uses a customized class loader and supports heterogeneous translation of parameterized class code. This approach provides substantial support for reflection—more than that of a homogeneous translation, though not the complete support described by Soloranzo and Alagic.

## 7 Conclusion

We've described how a restricted form of virtual typing can be added to a parameterized polymorphic implementation of Java in an efficient way. JL's *This* virtual type appears only in parametric types where it can always be statically resolved during type instantiation. Thorup [14] proposes that a similar construct be allowed in all classes at the cost of increased dynamic type checking.

Bruce [6] argues that a *ThisClass* construct based on the class type of *this* is inferior to *ThisType*. Bruce defines *ThisClass* as the class type of *this* and defines *ThisType* as the public interface of *this*. First, if *ThisClass* is used in an interface definition, the interface cannot be used as a type in standalone situations because no class binding is available. Second, calls to methods requiring an actual parameter of type *ThisClass* cannot be statically type checked because the type of the receiving object is not generally known at compile time.

JL's implementation avoids both of the problems described by Bruce by limiting the use of *This* to parameterized types. *This* is bound in every instantiation including parameterized interfaces in standalone situations. Since *This* is a type parameter that can appear only in parameterized types, type checking expressions involving *This* is the same as type checking expressions that involve any type parameter. See the Appendix for more discussion.

Agesen et al. [1] have proven the feasibility of a parametric polymorphic implementation of Java based on extending the class file format and the class loader. Agesen's approach does not change Java's type system or require modifications to current JVMs (or their security mechanisms). JL, though not yet implemented, can use the same approach and should be equally safe and feasible to implement.

## 8 References

- [1] O. Agesen, S. Freund and J. Mitchell. *Adding Type Parameterization to the Java Language*. OOPSLA 1997.
- [2] D. Batory's research group: <http://www.cs.utexas.edu/users/schwartz>.
- [3] D. Batory, B. Lofaso and Y. Smaragdakis. *JTS: Tools for Implementing Domain-Specific Languages*. International Conference of Software Reuse, Victoria, Canada, June 1998.
- [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. Bruce. *Increasing Java's expressiveness with ThisType and match-bounded polymorphism*. Technical Report, Williams College, 1997, <http://www.cs.williams.edu/~kim/README.html>.
- [6] K. Bruce, M. Odersky and P. Wadler. *A statically safe alternative to virtual types*. ECOOP 1998.
- [7] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys, Vol. 17, No. 4, December 1985.
- [8] R. Cardone, D. Batory and C. Lin. *Extending Java to Support Component-Based Programming*. Technical Report TR2000-11, Computer Sciences Department, University of Texas at Austin, 2000.
- [9] R. Cardone and C. Lin. *Comparing Frameworks and Layered Refinement*. Submitted for review, <http://www.cs.utexas.edu/users/richcar/JavaLayers.html>.
- [10] K. Czarnecki and U. Eisenecker. *Generative Programming*, Addison-Wesley, 2000.
- [11] J. Solorzano and S. Alagic. *Parametric Polymorphism of Java: A Reflective Solution*. OOPSLA 1998.
- [12] Y. Smaragdakis and D. Batory. *Implementing Layered Designs with Mixin Layers*. ECOOP 1998.
- [13] C. Strachey. *Fundamental concepts in programming languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.
- [14] K. Thorup. *Genericity in Java with Virtual Types*. ECOOP 1997.

## 9 Appendix

Since the version of JL described in this document has not been implemented yet, a number of design and implementation issues have been postponed. Below is a brief list of items that need further investigation.

### 1. Static versus dynamic type checking.

Treated as a type parameter, the *This* virtual type should present no special type checking difficulties when compared to explicitly defined type parameters. An analysis still needs to be performed to determine if and when dynamic type checks need to be inserted into instantiated code.

### 2. Improved canonical notation.

There's got to be a better notation for expressing parametric types with self-referential *This*-bindings. For example, `Node1<Node1<>>` should be written something like `Node1<#>`, where `#` stands for the class of the type being instantiated.

### 3. Constraints on explicitly specified *This*-Bindings.

A tighter specification would limit the classes that could explicitly be specified as a *This*-binding for a parametric class to only those that *match* the parameterized class. This idea of matching is like Bruce's matching except that it is only used to constrain the binding of *This*; it does not affect the relation between subclassing and subtyping. Matching is a weaker concept than subtyping. If class C is a subclass of class D, then C also matches D. If class E is a subclass of class F and if E had the same *This*-binding as F, then E matches F.

The implicit *This*-binding algorithm already binds only matching classes to parametric classes. Matching is useful in restricting explicit *This*-bindings to classes with some notion of conforming signatures. Static type checking of *This* within a class definition is beneficial because *This* doesn't have to be treated as a completely unconstrained type parameter. More importantly, this limited form of matching seems to be worthwhile because it restricts *This* to types that have some relationship to the class in which it appears. Programmers should appreciate the stronger semantics, though the details have yet to be worked out.

An even simpler approach would be to disallow explicit *This*-bindings on parametric class instantiations altogether. This would by default guarantee the above notion of matching with respect to *This*.