MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java

Curtis Clifton and Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
+1 515 294 1580
{cclifton, leavens}@cs.iastate.edu

Craig Chambers and Todd Millstein
Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 685 2094
{chambers, todd}@cs.washington.edu

ABSTRACT

We present MultiJava, a backward-compatible extension to Java supporting *open classes* and *symmetric multiple dispatch*. Open classes allow one to add to the set of methods that an existing class supports without creating distinct subclasses or editing existing code. Unlike the "Visitor" design pattern, open classes do not require advance planning, and open classes preserve the ability to add new subclasses modularly and safely. Multiple dispatch offers several well-known advantages over the single dispatching of conventional object-oriented languages, including a simple solution to some kinds of "binary method" problems. MultiJava's multiple dispatch retains Java's existing class-based encapsulation properties. We adapt previous theoretical work to allow compilation units to be statically typechecked modularly and safely, ruling out any link-time or run-time type errors. We also present a novel compilation scheme that operates modularly and incurs performance overhead only where open classes or multiple dispatching are actually used.

1. INTRODUCTION

In this paper we introduce MultiJava, a backward-compatible extension to Java™ [Gosling *et al.* 00, Arnold & Gosling 98] that supports *open classes* and *symmetric multiple dispatch*. An open class is one to which new methods can be added without editing the class directly [Chambers 98, Millstein & Chambers 99]. An open class allows clients to customize their interface to the needs of the client's application. Unlike customization through subclasses, in-place extension of classes does not require existing code referencing the class to be changed to use the new subclass instead. The "Visitor" design pattern [Gamma *et al*. 95, pp 331-344] also is intended to allow new client-specific operations to be added to an existing family of classes, but unlike open classes, the Visitor pattern requires class implementors to plan ahead and build infrastructure in the class with which clients can indirectly add behavior to the class. Moreover, unlike open classes, use of the Visitor pattern makes it difficult to add new subclasses modularly, since the existing Visitor infrastructure must be edited to account for the new subclasses. Open classes can be used to organize "cross-cutting" operations separately from the classes to which they belong, a key feature of aspect-oriented programming [Kiczales et al. 97]. With open classes, object-oriented languages can support the addition of both new subclasses and new methods to existing classes, relieving the tension that has been observed by others [Cook 90, Odersky & Wadler 97, Findler & Flatt 98] between these forms of extension.

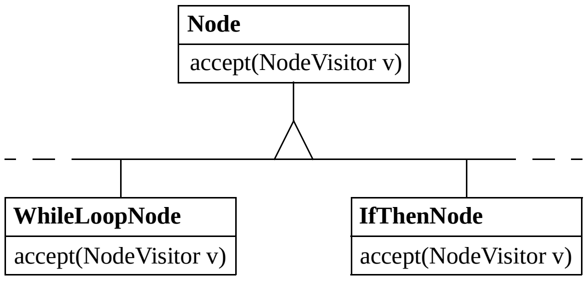
Multiple dispatch, found in Common Lisp [Steele 90, Paepcke 93], Dylan [Shalit 97, Feinberg et al. 97], and Cecil [Chambers 92, Chambers 95], allows the method invoked by a message send to depend on the run-time classes of any subset of the argument objects. A method that takes advantage of the multiple dispatch mechanism is called a *multimethod*. In contrast, *single dispatch*, found in C++, Smalltalk, and Java, selects the method invoked by a message send based on the run-time class of only the distinguished receiver argument. In C++ and Java, the static types of the arguments influence method selection via static overload resolution; the dynamic types of the arguments are not involved in method dispatch. Multimethods provide a more uniform and expressive approach to overload resolution. For example, they support safe covariant overriding in the face of subtype polymorphism, providing a natural solution to the "binary method" problem [Bruce et al. 95].

Multiple dispatch is *symmetric* if the rules for method lookup treat all dispatched arguments identically. *Asymmetric* multiple dispatch typically uses lexicographic ordering, where earlier arguments are more important; a variant of this approach selects methods based partly on the textual ordering of their declarations. We believe that symmetric multiple dispatch is more intuitive and less error-prone, reporting possible ambiguities rather than silently resolving them in potentially unexpected ways. Symmetric multiple dispatch is used in Cecil, Dylan, Kea [Mugridge et al. 91], the λ&-calculus [Castagna et al. 92, Castagna 97], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98].

A major obstacle to adding symmetric multimethods to an existing statically-typed programming language has been their modularity problem [Cook 90]: independently-developed modules, which typecheck in isolation, may cause type errors when combined. In contrast, object-oriented languages without multimethods do not suffer from this problem; for example, in Java, one can safely typecheck each compilation unit in isolation. Because of the multimethod modularity problem, previous work on adding multimethods to an existing statically-typed object-oriented language has either forced global typechecking [Leavens &

To be presented at the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00), Minneapolis, MN, October 2000.

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```



```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

Figure 1: Java code for some participants in the Visitor design pattern

Millstein 98] or has employed asymmetric multiple dispatch in order to ensure modularity [Boyland & Castagna 97]. Open classes can suffer from a similar modularity problem, if two unrelated clients each add the same method to the same class.

Our MultiJava language supports both open classes and symmetric multiple dispatch while retaining Java's modular encapsulation, typechecking, and compilation model. In particular, no new link-time or run-time typechecking or compilation needs to be performed. We achieve this goal by adapting previous work on modular static typechecking for open classes and multimethods in Dubious, a small multimethod-based core language [Millstein & Chambers 99]. One of our contributions is the extension of this previous theoretical result to the much larger, more complicated, and more practical Java language. A second contribution is a new compilation scheme for open classes and multimethods that is modular (each class or class extension can be compiled separately) and efficient (additional run-time cost is incurred only when multimethods or class extension methods are actually invoked). MultiJava is a conservative extension to Java: existing Java programs are legal MultiJava programs and have the same meaning. MultiJava retains backward-compatibility and interoperability with existing Java source and bytecode.

In the next two sections we present MultiJava's support for open classes and multiple dispatch. In Section 4 we more precisely describe MultiJava's modular static type system, and in Section 5 we outline the compilation of MultiJava source code into Java bytecode. In Section 6 we discuss an alternative language design for adding multiple dispatching to Java. Section 7 discusses related work and Section 8 concludes with several avenues for future work.

2. OPEN CLASSES

2.1 Motivation

Java allows a new subclass to be added to an existing class in a modular way—without requiring any modifications to existing code. However, Java (along with all other single-dispatch languages of which we are aware) does not allow a new method to be added to an existing class in a modular way. Instead, the programmer is forced to add the new method directly to the associated class declaration, and then to retypecheck and recompile the class. Of course, this requires access to the class's source code. Additionally, the new operation is then visible to all programs that use the class. Adding operations that are specific to a particular program can thus pollute the interface of the modified class, resulting in pressure to keep such ancillary operations out of the class. However, leaving an operation out of the class forfeits the benefits of run-time dispatching on different subclasses of the class, and it causes such "outside" methods to be invoked differently from methods declared inside the class declaration.

One potential approach to adding a new method to an existing class modularly is to add the new method in a new subclass of the class. This does not require modification of the original class declaration and allows new code to create instances of the new subclass and then access the new operation on subclass instances. However, it does not work if existing code already creates instances of the old class, or if a persistent database of old class instances is present. In such cases, non-modular conversions of existing code and databases to use the new subclass would be required, largely defeating the purpose of introducing the subclass in the first place. This approach will also work poorly if new subclass objects are passed to and later returned from existing code, since the existing code will return something whose static type is the old class, requiring an explicit downcast in order to access the new operation.

A second approach is to use the Visitor design pattern, which was developed specifically to address the problem of adding new functionality to existing classes in a modular way. The basic idea is to reify each operation into a class, thereby allowing operations to be structured in their own hierarchy.

For example, consider the `Node` class hierarchy in Figure 1. Instances of these classes are used by a compiler to form abstract

```
TypeDeclaration:                                                        // §7.6
    ExternalMethodDeclaration
    …
ExternalMethodDeclaration:                                              // new
    ExternalMethodHeader  MethodBody
ExternalMethodHeader:                                                   // new
    MethodModifiers_opt  ResultType  ExternalMethodDeclarator  Throws_opt
ExternalMethodDeclarator:                                               // new
    ClassType  .  Identifier  (  FormalParameterList_opt  )
```

Figure 2: Syntax extensions for MultiJava open classes

The grammar extends the Java syntax given in *The Java Language Specification* [Gosling, *et al.* 00, §2.4]. For standard Java nonterminals we just list the new productions for MultiJava and indicate the existence of the other productions with an ellipses (…). Existing Java nonterminals are annotated with the pertinent section numbers from *The Java Language Specification*.

syntax trees. The compiler might perform several operations on these abstract syntax trees, such as typechecking and code generation. These operations are structured in their own class hierarchy, each operation becoming a subclass of an abstract `NodeVisitor` class. The client of an operation on nodes invokes the `accept` method of a node, passing a `NodeVisitor` instance representing the operation to perform:

```
rootNode.accept(new TypeCheckingVisitor(...))
```

The `accept` method of each kind of node then uses double-dispatching [Ingalls 86] to invoke the visitor object's method appropriate for that type of node.

The main advantage of the Visitor pattern is that new operations can be added modularly, without needing to edit any of the `Node` subclasses: the programmer simply defines a new `NodeVisitor` subclass containing methods for visiting each class in the `Node` hierarchy. However, use of the Visitor pattern brings several drawbacks, including the following, listed in increasing importance:

- The stylized double-dispatching code is tedious to write and prone to error.
- The need for the Visitor pattern must be anticipated ahead of time, when the `Node` class is first implemented. For example, had the `Node` hierarchy not been written with an `accept` method, which allows visits from the `NodeVisitor` hierarchy, it would not have been possible to add typechecking functionality in a modular way. Even with the `accept` method included, only visitors that require no additional arguments and that return no results can be programmed in a natural way; unanticipated arguments or results can be handled only clumsily through state stored in the `NodeVisitor` subclass instance.
- Although the Visitor pattern allows the addition of new operations modularly, in so doing it gives up the ability to add new subclasses to existing `Node` classes in a modular way. For example, if a new `Node` subclass were introduced, the `NodeVisitor` class and all subclasses would have to be modified to contain a method for visiting the new kind of node. Proposals have been advanced for dealing with this well-known limitation [Martin 98, Nordberg 98, Vlissides 99], but they suffer from additional complexity (in the form of hand-coded type-cases and more complex class hierarchies) that make them even more difficult and error-prone to use.

2.2 Open Classes in MultiJava

2.2.1 Declaring and Invoking Top-Level Methods

The open class feature of MultiJava allows a programmer to add new methods to existing classes without modifying existing code and without breaking the encapsulation properties of Java. The key new language feature involved is the top-level method declaration, whose syntax is specified in Figure 2. Using top-level methods, the functionality of the typechecking visitor from Figure 1 can be written as follows:

```
// compilation unit "typeCheck"
package oopsla.examples;

// Methods for typechecking
public boolean Node.typeCheck()
  { /* ... */ }
public boolean WhileLoopNode.typeCheck()
  { /* ... */ }
public boolean IfThenNode.typeCheck()
  { /* ... */ }
```

A program may contain several top-level method declarations that add methods to the same class. As in Java, the bodies of top-level methods may use **`this`** to reference the receiver object.

Clients may invoke top-level methods exactly as they would use the class's original methods. For example, the `typeCheck` method of `rootNode` is invoked as follows:

```
rootNode.typeCheck()
```

where `rootNode` is an instance of `Node` or a subclass. This is allowed even if `rootNode` was created by code that did not have access to the `typeCheck` methods, or was retrieved from a persistent database. Code can create and manipulate instances of classes without being aware of all top-level methods that may have been added to the classes; only code wishing to invoke a particular top-level method needs to be aware of its declaration.

2.2.2 Generic Functions, External and Internal

It is helpful at this point to define some technical terms.

Conceptually, one can think of each method in the program (whether top-level or declared within classes) as implicitly belonging to a *generic function*, which is a collection of methods consisting of a *top method* and all of the methods that (dynamically) override it. For example, the `typeCheck` top-level methods above introduce a single new generic function, providing implementations for three receiver classes. Each message send site invokes the methods of a particular, statically determined generic function.

More precisely, given a method declaration M_{sub} whose receiver is of class or interface T, if there is a method declaration M_{sup} of the same name, number of arguments, and static argument types as M_{sub} but whose receiver is of some proper supertype of T, then M_{sub} belongs to the same generic function as M_{sup}, hence M_{sub} overrides M_{sup}. Otherwise, M_{sub} is the top method of a new generic function. The top method may be abstract, for example if it is declared in an interface.

We say that a class S is a *subtype* of a type T (equivalently, T is a *supertype* of S) if one of the following holds:

- T is a class and S is either T or a subclass of T,
- T is an interface and S is a class that implements T, or
- T is an interface and S is an interface that extends T.

We say that S is a *proper subtype* of T if S is a subtype of T and S is not the same as T.

We call a method declared via the top-level method declaration syntax an *external method* if the class of its receiver is not declared in the same *compilation unit*. (A Java compilation unit corresponds to a single file in Java implementations based on file systems [Gosling *et al.* 00, §7.6].) All other methods are *internal*. Besides methods declared in class declarations, this includes methods declared via the top-level method declaration syntax whose receiver class is declared in the same compilation unit. Calling such methods "internal" is sensible, since they can be added to the receiver's class declaration by the compiler.

Analogously, a generic function is external if its top method is external. All other generic functions are internal. Some methods of an external generic function can be internal methods (see Subsection 2.2.4).

2.2.3 Scoping of External Generic Functions

To invoke or override an external generic function, client code first imports the generic function using an extension of Java's existing import mechanism. For example,

```
import oopsla.examples.typeCheck;
```

will import the compilation unit *typeCheck* from the package `oopsla.examples`, which in this case declares the `typeCheck` generic function. Similarly

```
import oopsla.examples.*;
```

will implicitly import all the compilation units in the package `oopsla.examples`, which will make all types and generic functions in that package available for use. Each compilation unit implicitly imports all the generic functions in its package.

The explicit importation of external generic functions enables client code to manage the name spaces of the classes they manipulate. Only clients that import the *typeCheck* compilation unit will have the `typeCheck` operation in the interface to `Node`. Other clients will not have their interfaces to `Node` polluted with this generic function. Furthermore, a compilation unit that did not import the *typeCheck* compilation unit could declare its own `typeCheck` generic function without conflict.

Java allows at most one public type (class or interface) declaration in a compilation unit [Gosling *et al.* 00, §7.6].[1] This concession allows the implementation to find the file containing the code for a type based on its name. In MultiJava we extend this restriction in a natural way: each file may contain either one public type with associated internal methods, or the top method (and any number of overriding methods) of one public external generic function.

1. Java's restriction is somewhat more complex to account for its default access modifier, which gives access to all other classes in the package.

2.2.4 Subsuming the Visitor Pattern

A key benefit of open classes is that they obviate the need for the Visitor pattern infrastructure; the `Visitor` class hierarchy and `accept` methods of the `Node`s are now unnecessary. Instead, the client-specific operations to be performed can be written as top-level methods of `Node` and its subclasses, outside of the class declarations. Unlike with the Visitor pattern, there is no need to plan ahead for adding the new operations, i.e., new external generic functions. Each new external generic function can define its own argument types and result type, independently of other operations. More importantly, this idiom still allows new `Node` subclasses to be added to the program modularly, because there is no `Visitor` hierarchy that needs to be updated. For example, a new subclass of `Node` can be added without changing any existing code, as follows:

```
import oopsla.examples.Node;
import oopsla.examples.typeCheck;
public class DoUntilNode extends Node {
  /* ... */
  public boolean typeCheck()
    { /* ... */ }
}
```

MultiJava extends Java's notion of method inheritance to open classes. A client of `DoUntilNode` can invoke any visible `Node` method on an instance of `DoUntilNode`, regardless of whether that method was visible in `DoUntilNode`'s compilation unit.

A subclass also can override any visible inherited (internal or external) methods, as in the example above. (This example also illustrates that regular internal methods can be added to external generic functions.)

The ability to write external methods gives programmers more flexibility in organizing their code. For example, the original three `typeCheck` methods can all be put in a single file separate from the compilation units defining the classes of the `Node` hierarchy. Open classes also allow new methods to be added to an existing class even if the source code of the class is not available, for example if the class is in a Java library. New methods can even be added to a **`final`** class without violating the property that the class has no subclasses.

2.2.5 Encapsulation

MultiJava retains the same encapsulation properties as Java [Gosling *et al.* 00, §6.6]. All Java privileged access modifiers are allowed for external methods. For example, a helper method for a public external method may be declared **`private`** and included in the same compilation unit as the public method. These modifiers have the usual meaning for methods, with the exception that a private external method may only be invoked or overridden from within the compilation unit in which it is declared. (This differs from Java because the context of an external method is a compilation unit instead of a class.)[2]

Further, an external method may access:

- **`public`** members of its receiver class, and
- non-**`private`** members of its receiver class if the external method is in the same package as that class.

2. In Java, a protected method can be overridden within subclasses of its receiver class. In MultiJava one can also define protected external methods; these can be overridden both in subclasses and also within the compilation unit in which they are introduced.

All other access to receiver class members is prohibited. In particular, an external method does not have access to the private members of its receiver class. A top-level internal method has the same access privileges as a regular Java method, including the ability to access private members of its receiver class.

2.2.6 Restrictions for Modular Typechecking

As a consequence of MultiJava's modular typechecking scheme discussed in Section 4, external methods may not be annotated as **`abstract`**, nor can they be added to interfaces. Since concrete subclasses of the extended abstract class or interface can be declared in other compilation units, without knowledge of such an abstract external method and vice versa, purely modular typechecking could not guarantee that the external generic function was implemented for all concrete subclasses of the abstract class or interface.

A second consequence of modular typechecking is that a top-level method must either belong to a generic function whose top method is in the same compilation unit, or it must be an internal method. Without this restriction, it would be possible for independent compilation units to declare top-level methods in the same generic function with the same receiver class, leading to a clash.

Both of these restrictions are discussed in Section 4.

2.2.7 Other Class Extensions

MultiJava currently allows only instance (non-**`static`**) methods to be added to existing classes. However, it should be straightforward to extend our work to allow top-level static methods and even top-level static fields. Top-level instance fields and top-level instance constructors would require more significant extensions to our compilation strategy.

3. MULTIPLE DISPATCH

3.1 Motivation

In Java, the method invoked by a message can depend on the run-time class of the receiver object, but it cannot depend on the run-time classes of any other arguments. In some situations, this restriction is unnatural and limiting. One common example involves *binary methods*. A binary method is a method that operates on two or more objects of the same type [Bruce *et al.* 95]. In the `Shape` class below, the method for checking whether two shapes intersect is a binary method.

```
public class Shape {
  /* ... */
  public boolean intersect(Shape s) {
    /* ... */
  }
}
```

Now suppose that one wishes to create a class `Rectangle` as a subclass of `Shape`. When comparing two rectangles, one can use a more efficient intersection algorithm than when comparing arbitrary shapes. The first way one might attempt to add this functionality in a Java program is as follows:

```
public class Rectangle extends Shape {
  /* ... */
  public boolean intersect(Rectangle r) {
    /* efficient code for two Rectangles */
  }
}
```

Unfortunately, this does not provide the desired semantics. In particular, the new intersection method cannot be safely considered to override the original intersection method, because it violates the standard contravariant typechecking rule for functions [Cardelli 88]: the argument type cannot safely be changed to a subtype in the overriding method. Suppose the new method were considered to override the intersection method from class `Shape`. Then a method invocation `s1.intersect(s2)` in Java would invoke the overriding method whenever `s1` is an instance of `Rectangle`, regardless of the run-time class of `s2`. Therefore, it would be possible to invoke the `Rectangle` intersection method when `s2` is an arbitrary `Shape`, even though the method expects its argument to be another `Rectangle`. This could cause a run-time type error, for example if `Rectangle`'s method tries to access a field in its argument `r` that is not inherited from `Shape`.

To handle this problem, Java, like C++, considers `Rectangle`'s `intersect` method to *statically overload* `Shape`'s method. Statically overloaded methods belong to distinct generic functions, just as if the methods had different names. Java uses the name, number of arguments, and static argument types of a message send to statically determine which generic function is invoked at each message send site. In our example, because of the different static argument types, the two `intersect` methods belong to different generic functions, and Java determines statically which generic function is invoked for each `intersect` message send site based on the *static* type of the message argument expression. For example, consider the following client code:

```
Rectangle r1, r2;
Shape s1, s2;
boolean b1, b2, b3, b4;
r1 = new Rectangle( /* ... */ );
r2 = new Rectangle( /* ... */ );
s1 = r1;
s2 = r2;
b1 = r1.intersect(r2);
b2 = r1.intersect(s2);
b3 = s1.intersect(r2);
b4 = s1.intersect(s2);
```

Although the objects passed as arguments in the four `intersect` message sends above are identical, these message sends do not all invoke the same method. In fact, only the first message send will invoke the `Rectangle` intersection method. The other three messages will invoke the `Shape` intersection method, because the static types of these arguments cause Java to bind the messages to the generic function introduced by the `Shape intersect` method. Likewise, the first message is statically bound to the generic function introduced by the `Rectangle intersect` method.

In Java, one can solve this problem by performing explicit run-time type tests and associated casts; we call this coding pattern a *type-case*. For example, one could implement the `Rectangle` intersection method as follows:

```
public class Rectangle extends Shape {
  /* ... */
  public boolean intersect(Shape s) {
    if (s instanceof Rectangle) {
      Rectangle r = (Rectangle) s;
      // efficient code for two Rectangles
    } else {
      super.intersect(s);
    }
  }
}
```

This version of the `Rectangle` intersection method has the desired semantics. In addition, since it takes an argument of type `Shape`, this method can safely override `Shape`'s `intersect` method, and is part of the same generic function. All message sends

```
FormalParameter:                         // §8.4.1
    Type @ ClassType VariableDeclaratorId
    …
```

Figure 3: Syntax extensions for MultiJava multimethods

in the example client code above will now invoke the `Rectangle` `intersect` method.

However, this "improved" code has several problems. First, the programmer is explicitly coding the search for what intersection algorithm to execute, which can be tedious and error-prone. In addition, such code is not easily extensible. For example, suppose a `Triangle` subclass of `Shape` is added to the program. If special intersection behavior is required of a `Rectangle` and a `Triangle`, the above method must be modified to add the new case. In general, whenever a new `Shape` subclass is added, the type-case of each existing binary method of each existing `Shape` subclass may need to be modified to add a new case for the new `Shape` subclass.

A related solution to the binary method problem in Java is the use of double-dispatching, as in the `accept` methods of the Visitor pattern (see Figure 1). With this technique, instead of using an explicit **`instanceof`** test to find out the run-time type of the argument `s`, as in the above example, this information is obtained by performing a second message send. This message is sent to the argument `s`, but with the name of the message encoding the dynamic class of the original receiver. Double-dispatching avoids the need for the `intersect` method of every `Shape` subclass to include an explicit type-case over all the possible argument shapes, and it reuses the language's built-in method dispatching mechanism in place of user-written type-cases. However, double-dispatching is even more tedious to implement by hand than type-casing. Finally, double-dispatching is still not completely modular, since it requires at least the root class (`Shape` in our example) to be modified whenever a new subclass is to be added.

3.2 Multiple Dispatch in MultiJava

3.2.1 Declaring Multimethods

In part to provide a clean and modular solution to the binary method problem, MultiJava allows programmers to write multimethods, which are methods that can dynamically dispatch on other arguments in addition to the receiver object. The syntax of our multimethod extension is specified in Figure 3. Using multimethods, the definition of the `Rectangle` class can be changed to the following:

```
public class Rectangle extends Shape {
  /* ... */
  public boolean
    intersect(Shape@Rectangle r) {
    /* efficient code for two Rectangles */
  }
}
```

This code is identical to the first solution attempt presented in Subsection 3.1, except that the type declaration of the formal parameter `r` is `Shape@Rectangle` instead of simply `Rectangle`. The `Shape` part denotes the *static* type of the argument `r`. Consequently, the revised `Rectangle` `intersect` method belongs to the same generic function as the `Shape` `intersect` method, because the name, number of arguments, and (static) argument types match. The `@Rectangle` part indicates that we wish to dynamically dispatch on the formal parameter `r`, in addition to the receiver. As with standard Java, the receiver is always dispatched upon. So this `intersect` method will be invoked only if the dynamic class of the receiver is `Rectangle` or a subclass (as with regular Java) *and* the dynamic class of the argument `r` is `Rectangle` or a subclass.

3.2.2 Message Dispatch Semantics

In a formal parameter declaration, the class after an `@` symbol is referred to as the *explicit specializer* of the formal. For a given method M, its *tuple of specializers* $(S_0,...,S_n)$ is such that S_0 is M's receiver type and, for $i \in \{1..n\}$, if M has an explicit specializer, U_i, at the ith position, then S_i is U_i, otherwise S_i is the static type of the ith argument. Thus the `Shape` class's `intersect` method has the tuple of specializers (`Shape`, `Shape`) while the `Rectangle` class's method has (`Rectangle`, `Rectangle`).

The semantics of message dispatch in MultiJava is as follows. For a message send $E_0.I(E_1,...,E_n)$, we evaluate each E_i to some value v_i, extract the methods in the generic function being invoked (determined statically based on the generic functions in scope named I that are appropriate for the static types of the E_i expressions), and then select and invoke the *most-specific* such method *applicable* to the arguments $(v_0,...,v_n)$. Let $(C_0,...,C_n)$ be the dynamic types of $(v_0,...,v_n)$; if v_i is not an object, let C_i be its static type. We extend the subtype relation defined in Subsection 2.2.2 to primitive types, which are subtypes of themselves only. A method with tuple of specializers $(S_0,...,S_n)$ is *applicable* to $(v_0,...,v_n)$ if $(C_0,...,C_n)$ *pointwise subtypes* from $(S_0,...,S_n)$ (that is, for each i, C_i is a subtype of S_i). The *most-specific* applicable method is the unique applicable method whose tuple of specializers $(S_0,...,S_n)$ pointwise subtypes from the tuple of specializers of every applicable method. If there are no applicable methods, a *message-not-understood* error occurs; we say a generic function is *incomplete* if it can cause message-not-understood errors when invoked. If there are applicable methods but no unique most-specific one, a *message-ambiguous* error occurs; we say a generic function is *ambiguous* if it can cause message-ambiguous errors when invoked. (Static typechecking, described in Section 4, can always detect and reject generic functions that are potentially incomplete or ambiguous.)

Given this dispatching semantics, the above code indeed solves the binary method problem. For example, consider an invocation `s1.intersect(s2)`, where `s1` and `s2` have static type `Shape`. If at run time both arguments are instances of `Rectangle` (or a subclass of `Rectangle`), then both the `Shape` and `Rectangle` `intersect` methods are applicable. Of these applicable methods, the `Rectangle` method is the most specific, and therefore it will be selected and invoked. Otherwise, only the `Shape` method is applicable, and it will therefore be invoked.

MultiJava's dispatching semantics naturally generalizes Java's dispatching semantics. If a MultiJava program uses no `@` argument specializers, then dispatching occurs only on the receiver and the behavior of the program is exactly as in regular Java. The semantics of both dynamic dispatching and static overloading are unchanged. The addition of `@` argument specializers extends Java's normal dynamic dispatching semantics to these additional arguments.

3.2.3 Mixing Methods with Multimethods

Any subset of a method's arguments can be specialized. A class can declare several methods with the same name and static argument types, provided they have different argument specializers and no ambiguities arise. For example, a `Circle` class could be defined with a selection of intersection methods:

```
public class Circle extends Shape {
  /* ... */
  public boolean
    intersect(Shape s) {
    /* code for a Circle against any Shape */
  }
  public boolean
    intersect(Shape@Rectangle r) {
    /* efficient code against a Rectangle */
  }
  public boolean
    intersect(Shape@Circle c) {
    /* very efficient code for two Circles */
  }
}
```

All these methods have static argument type `Shape`, so they all are in the same generic function (introduced by the `intersect` method in the `Shape` class). However, they have different combinations of specializers, causing them to apply to different run-time circumstances. For example, consider again the `s1.intersect(s2)` invocation, where `s1` and `s2` have static type `Shape`. If at run time both arguments are instances of `Circle`, then the first and third of these methods are applicable, along with the `Shape` class's default `intersect` method. The third `Circle` method is pointwise most specific, so it is invoked. If `s1` is a `Circle` but `s2` is a `Triangle`, then only the first `Circle` method and the `Shape` method are applicable, and the first `Circle` method is invoked. If `s1` is a `Rectangle` and `s2` is a `Circle`, then only the `Shape intersect` method remains applicable.

In general, a generic function can include methods that specialize on different subsets of arguments, as long as it is not ambiguous. (Ambiguity detection is discussed in Section 4.) Invocations of generic functions use regular Java message syntax, and do not depend on which arguments are specialized. A regular Java method can be overridden in a subclass with a multimethod, without modifying the overridden class or any invocations of the method.

3.2.4 Super Sends

Java's **`super`** construct allows a method to invoke the method it directly overrides. Java also allows such a *super send* to invoke a method in a different generic function, if the name of the message is different than the sender's name or if the arguments differ in number or static type from the formal parameters of the sender.

For MultiJava, a multimethod may override some other method in the same compilation unit. For example, the third `Circle intersect` method above overrides the first `Circle intersect` method. MultiJava super sends should thus be able to walk up the chain of overriding methods, even within the same compilation unit. However, MultiJava should also retain backward compatibility with Java in that **`super`** should be able to invoke a method from a different generic function than the sender.

Our solution is first to statically identify for each super send whether it will invoke a method from the same generic function as the sender. We do this based on the name of the message and the number and static types of the arguments. If the target generic function is the same as the sender's, then the semantics of the super send is the same as the semantics of a regular MultiJava message send, except that the set of applicable methods is filtered to include only methods that are overridden (directly or indirectly) by the sending method. If, on the other hand, the target generic function is different from the sender's, then as in Java, MultiJava filters the set of applicable methods to include only methods declared for or inherited by the sender's immediate superclass.

To illustrate these semantics, consider an implementation of the third `Circle intersect` method that contains a super send to `intersect` with the same arguments:

```
public boolean intersect(Shape@Circle c) {
  ... super.intersect(c) ...
}
```

This invocation is known (statically) to invoke a method in the same generic function as the sender's. Consequently, the set of applicable methods are those whose receiver and argument specializer types are pointwise supertypes of the run-time receiver and argument classes, filtered to contain just those that are overridden by this `intersect` method. In this case, the filtered applicable methods are the first `Circle intersect` method and the `intersect` method from `Shape`. The unique, most-specific applicable method is the first `Circle intersect` method—so it is invoked. If that method itself contains a super send of the same form, then the only filtered applicable method would be `Shape`'s. A super send of the same form in `Shape`'s `intersect` method would lead to a static type error, as there would be no applicable methods.

Now consider an alternative implementation of the third `Circle intersect` method containing a different super send:

```
public boolean intersect(Shape@Circle c) {
  ... super.specialIntersect(c) ...
}
```

This invocation is directed to a different generic function than the sender's, and so the set of applicable methods consists of only those applicable methods of the `specialIntersect` generic function that are declared in or inherited by `Shape`, `Circle`'s immediate superclass.

3.2.5 Other Uses of Multimethods

While binary methods are a commonly occurring situation where multimethods are valuable, other situations can benefit from multiple dispatching as well. For one example, consider a `displayOn` generic function defined over shapes and output devices. Default `displayOn` algorithms would be provided for each shape, independent of the output device. However, certain combinations of a shape and an output device might allow more efficient algorithms, for instance if the device provides hardware support for rendering the shape. To implement this generic function, the `Shape` class could introduce a `displayOn` method:

```
public class Shape {
  /* ... */
  public void displayOn(OutputDevice d) {
      /* default display of shape */
  }
}
```

Each subclass of `Shape` would be able to provide additional overriding `displayOn` multimethods for particular kinds of output devices. For example, the `Rectangle` class might provide a few `displayOn` multimethods:

```
public class Rectangle extends Shape {
  /* ... */
  public void
    displayOn(OutputDevice d) {
      /* default display of rectangle */
  }
  public void
    displayOn(OutputDevice@XWindow d) {
      /* special display of rectangle on X Windows */
  }
  public void
    displayOn(OutputDevice@FastHardware d) {
      /* fast display of rectangle using hardware support */
  }
}
```

Top-level methods added to open classes can also be multimethods. For example, the above `displayOn` generic function could be implemented as an external generic function.

3.2.6 Restrictions for Modular Typechecking

When a multimethod is external all the restrictions for open classes apply; for example, external multimethods cannot be abstract.

Whether a multimethod is internal or external, default implementations must be provided for arguments that have non-concrete static types. For example, assuming that the `OutputDevice` class above is abstract, the first `displayOn` method for the `Rectangle` class provides this default implementation for the argument tuple (`Rectangle`, `OutputDevice`). We discuss this restriction further in Subsection 4.2.2.

4. TYPECHECKING

In this section we describe how to extend Java's static type system to accommodate MultiJava's extensions. We present the overall structure of our modular type system in Subsection 4.1. In Subsection 4.2 we describe several challenges that open classes and multimethods pose for modular typechecking, and we discuss the restrictions we impose in MultiJava to meet those challenges.

4.1 Overall Approach

The MultiJava type system ensures statically that no message-not-understood or message-ambiguous errors can occur at run time. Ruling out these errors involves complementary *client-side checking* of message sends and *implementation-side checking* of methods [Chambers & Leavens 95]. We begin by describing what we mean by modular typechecking, and then discuss the two kinds of checks.

4.1.1 Modular Typechecking

Modular typechecking requires that each compilation unit can be successfully typechecked only considering static type information from the compilation units that it imports. If all compilation units separately pass their static typechecks, then every combination of compilation units (that pass the regular Java link-time checks) is safe: there is no possibility of a message-not-understood or message-ambiguous error at run time.

We say that a type is *visible* in a compilation unit U if it is declared in or referred to in U, or if the type is a primitive type. A tuple of types is visible if each component type is visible. A method is *visible* in a compilation unit U if it is declared in U, declared in a type T that is visible in U, or is an external method declared in a compilation unit that is imported by U. A modular typechecking strategy only needs to consider visible types and visible methods to determine whether a compilation unit is type-correct.

4.1.2 Client-side Typechecking

Client-side checks are local checks for type correctness of each message send expression. For each message send expression $E_0.I(E_1,...,E_n)$ in the program, let T_i be the static type of E_i. Then there must exist a unique generic function in scope named I whose top method has a tuple of argument types $(T_0',...,T_n')$ that is a pointwise supertype of $(T_0,...,T_n)$. This check is already performed in standard Java. In our extension, however, external generic functions that are imported must be checked along with regular class and interface declarations.

For a send whose receiver is **`super`**, the typechecker must additionally ensure that there exists a unique, most-specific, non-abstract method invoked by the send. This check extends the checking on super sends that Java performs already.

4.1.3 Implementation-side Typechecking

Implementation-side checks ensure that each generic function is fully and unambiguously implemented. These checks have two parts.

4.1.3.1 Checks on Individual Method Declarations

The first part applies to each method declaration M in isolation:

- For each of M's explicit specializers, S, the associated static type must be a proper supertype of S, and S must be a class.
- If M is an overriding method then its privileged access modifiers must satisfy the following:
  - If M overrides a method M_2 of the same compilation unit with the same receiver, then M must have the same access level as M_2.
  - If M belongs to an external generic function, then it must have the same access level as the generic function's top method.
  - Otherwise, the standard Java rules for privileged access and overriding apply [Gosling *et al.* 00, §8.4.6.3].

Requiring an explicit specializer to be a proper subtype of the associated static type ensures that the specializer will affect dynamic dispatching. If the specializer were a supertype of the associated static type, then the specializer would be applicable to every legal message send of the generic function, which is equivalent to not specializing at that argument position. Furthermore, if the specializer were unrelated to the associated static type, then the specializer would be applicable to no legal message sends of the generic function, so the method would never be invoked. The explicit specializers are required to be classes rather than interfaces because the form of multiple inheritance supported by interfaces can create ambiguities that elude modular static detection [Millstein & Chambers 99].

The restrictions on privileged access level for overriding methods are intended to simplify the compilation scheme. For example, the restrictions ensure that the set of methods in a compilation unit that are part of the same generic function have the same privileged access modifiers. Therefore, these methods can be compiled into a single Java method. We leave exploring ways to relax these restrictions as future work.

4.1.3.2 Checks on Entire Generic Functions

The second part of the implementation-side checks treats all the visible multimethods in a visible generic function as a group. Consider a generic function whose top method has argument types $(T_0,...,T_n)$. A tuple of types $(C_0,...,C_n)$ is a *legal argument tuple* of the generic function if $(T_0,...,T_n)$ is a pointwise supertype of $(C_0,...,C_n)$ and each C_i is concrete. We say that a type is *concrete* if it is a primitive type or if it is a class that is not declared **abstract**. Abstract classes and interfaces are *non-concrete*.The checks are that for each visible generic function, each visible legal argument tuple has a visible, most-specific applicable method to invoke. This part of implementation-side typechecking is critical for ruling out ambiguities between multimethods and for ensuring that abstract top methods are overridden with non-abstract methods for all combinations of concrete arguments.

For example, consider implementation-side checks on the `intersect` generic function, from the perspective of a compilation unit containing only the `Rectangle` class as defined in Subsection 3.2. From this compilation unit, `Shape` and `Rectangle` are the only visible `Shape` subclasses (`Circle` is not visible, because it is not referenced by the `Rectangle` class). The `intersect` generic function is visible, as are two `intersect` methods (one each in `Shape` and `Rectangle`). There are four visible legal argument tuples: all pairs of `Shapes` and `Rectangles`. The `intersect` method in class `Rectangle` is the most specific applicable method for the (`Rectangle`, `Rectangle`) tuple while the `intersect` method in class `Shape` is the most specific applicable method for the other three tuples. Conceptually, this checking involves an enumeration of all combinations of visible legal argument tuples, but more efficient algorithms exist that only check the "interesting" subset of tuples [Chambers & Leavens 95, Castagna 97].

4.2 Restrictions for Modular Type Safety

Unfortunately, the typechecking approach described above can miss message-not-understood or message-ambiguous errors that may occur at run time, caused by interactions between unrelated compilation units [Millstein & Chambers 99]. In the rest of this subsection, we describe the kinds of errors that can occur, and explain the restrictions we impose in MultiJava to rule them out.

4.2.1 Abstract Classes and Open Classes

As mentioned previously in Subsection 2.2.6, abstract external methods can lead to message-not-understood errors. This is illustrated in Figure 4. The `JPEG` class is a concrete implementation of the abstract `Picture` class. The external method declaration in the *draw* compilation unit adds a new abstract method, `draw`, to the abstract `Picture` class. The *draw* compilation unit passes the implementation-side typechecks because the `JPEG` class is not visible. However, if a client ever invokes `draw` on a `JPEG`, a message-not-understood error will occur.

To rule out this problem, we impose restriction *R1*:

> (***R1***) Implementation-side typechecks of a local, external generic function must consider any non-local, non-concrete visible subtypes of its receiver type to be concrete at the receiver position.

A type or method is *local* if it is declared in the current compilation unit, and otherwise it is *non-local*. A generic function is *local* if its top method is local, and otherwise it is *non-local*.

In Figure 4, the external `draw` method in the compilation unit *draw* introduces a new generic function with the non-local, non-concrete receiver `Picture`. By restriction ***R1***, implementation-side typechecks must consider `Picture` to be concrete, thereby finding an incompleteness for the legal argument tuple (`Picture`). Therefore, the *draw* compilation unit must provide an implementation for drawing `Picture`s, which resolves the incompleteness for the unseen `JPEG` class.

```
// compilation unit "Picture"
package oopsla;
public abstract class Picture {
  /* ... no draw method ... */
}
// compilation unit "JPEG"
import oopsla.Picture;
public class JPEG extends Picture {
  /* ... no draw method ... */
}
// compilation unit "draw"
import oopsla.Picture;
public abstract void Picture.draw();
```

Figure 4: Incompleteness problem with abstract classes and open classes

```
// compilation unit "Picture"
package oopsla;
public abstract class Picture {
  public abstract boolean similar(Picture p);
}
// compilation unit "JPEG"
import oopsla.Picture;
public class JPEG extends Picture {
  public boolean similar(Picture@JPEG j)
    { /* ... */ }
}
// compilation unit "GIF"
import oopsla.Picture;
public class GIF extends Picture {
  public boolean similar(Picture@GIF g)
    { /* ... */ }
}
```

Figure 5: Incompleteness problem with abstract classes and multimethods

```
// compilation unit "intersect"
import oopsla.Shape;
import oopsla.Rectangle;
public boolean
 Shape.intersect(Shape@Rectangle r)
  { /* ... */ }
// compilation unit "Triangle"
import oopsla.Shape;
public class Triangle extends Shape {
  public boolean intersect(Shape s) {
    /* ... */ }
}
```

Figure 6: Ambiguity problem with unrestricted multimethods

As a consequence of restriction ***R1***, it is useless to declare an external method **abstract**, since the restriction will force the receiver class to be treated as concrete anyway. For the same reason, MultiJava cannot support open interfaces, i.e., the ability to add method signatures to interfaces.

4.2.2 Abstract Classes and Multimethods

Abstract classes coupled with multimethods can also lead to message-not-understood errors. Consider the example in Figure 5. Since the `Picture` class is declared **abstract**, it need not implement the `similar` method. Implementation-side checks of the *JPEG* compilation unit verify that the single visible legal argument tuple, (`JPEG`, `JPEG`), has a most-specific `similar` method, and similarly for the *GIF* compilation unit. However, at run time, a message-not-understood error will occur if the `similar` message is sent to one `JPEG` and one `GIF`.

To rule out this problem, we impose restriction ***R2***:

> (***R2***) For each non-receiver argument position, implementation-side typechecks of a generic function must consider all non-concrete visible subtypes of its static type to be concrete at that argument position.

In Figure 5, since `Picture` is abstract, by restriction ***R2*** implementation-side typechecks on the `similar` generic function from `JPEG`'s compilation unit must consider `Picture` to be concrete on the non-receiver argument position. Therefore, these checks will find an incompleteness for the legal argument tuple (`JPEG`, `Picture`), requiring the `JPEG` class to include a method handling this case, which therefore also handles the (`JPEG`, `GIF`) argument tuple. Similarly, the `GIF` class will be forced to add a `similar` method handling (`GIF`, `Picture`). In general, restriction ***R2*** forces the creation of method implementations to handle abstract classes on non-receiver arguments of multimethods. This ensures that appropriate method implementations exist to handle any unseen concrete subclasses of the abstract classes.

Restriction ***R1*** complements ***R2***, addressing the case of abstract classes at the receiver position. As in ***R2***, the existence of appropriate method implementations to handle the abstract classes is ensured. However, restriction ***R1*** applies only to external generic functions, so internal generic functions may safely use abstract classes in the receiver position. This permits all the uses of abstract classes and methods allowed by standard Java, as well as some uses with multimethods. For example, in Figure 5 the abstract `Picture` class may safely omit an implementation of the internal `similar` generic function.

4.2.3 Unrestricted Method Overriding

Message-ambiguous errors that elude static detection can occur if arbitrary methods can be added to a generic function by any compilation unit. These errors can occur without multiple dispatch (as mentioned in Subsection 2.2.6). In this section we give an example that uses multiple dispatch.

Consider the example in Figure 6, assuming the `Shape` class from Subsection 3.1 and `Rectangle` class from Subsection 3.2. The external method declaration in compilation unit *intersect* overrides the default `Shape intersect` method for arguments whose dynamic class is `Rectangle`. `Shape`s and `Rectangle`s are visible in the *intersect* compilation unit, and every pair of these classes has a most-specific applicable method. Similarly, `Shape`s and `Triangle`s are visible in the *Triangle* compilation unit, and its implementation-side checks also succeed. However, at run time, an `intersect` message send with one `Triangle` instance and one `Rectangle` instance will cause a message-ambiguous error to occur, because neither method in the example is more specific than the other.

One way to partially solve this problem is to break the symmetry of the dispatching semantics. For example, if we linearized the specificity of argument positions, comparing specializers lexicographically left-to-right (rather than pointwise) as is done in Common Lisp [Steele 90, Paepcke 93] and Polyglot [Agrawal et al. 91], then the method in *Triangle* would be strictly more specific than the method in *intersect*. However, one of our major design goals is to retain the symmetric multimethod dispatching semantics. Furthermore, unrestricted external methods would allow one to create two methods with identical type signatures; breaking the symmetry of dispatching cannot solve this part of the problem.

Our solution is to impose restriction ***R3***:

> (***R3***) An external method must belong to a local generic function.

In Figure 6, the external method declaration in the *intersect* compilation unit violates restriction ***R3***. In particular, the associated `intersect` method does not belong to a local generic function; the `intersect` generic function's top method is in the non-local `Shape` class. Therefore, by restriction ***R3***, the only legal location for the declaration of an `intersect` method with tuple of specializers (`Shape`, `Rectangle`) is within the same compilation unit as the `Shape` class. In that case, the method declaration and the `Rectangle` class would be visible to the *Triangle* compilation unit, which would therefore check for a most-specific applicable method for the argument tuple (`Triangle`, `Rectangle`), statically detecting the ambiguity. To resolve this ambiguity one must write a method that dispatches on the (`Triangle`, `Rectangle`) tuple.

As a result of restriction ***R3***, each method declaration *M* must be in the same compilation unit as either the receiver's class or the associated generic function's top method. In either case, any unseen method M_2 of the same generic function must have a different receiver than *M*, or M_2 would be in violation of restriction ***R3***. Therefore, method *M* cannot be ambiguous with any unseen method M_2, so the modular implementation-side typechecks are enough to rule out any potential ambiguities.

In Java, one can declare that a method is **`final`**, which prevents it from being overridden. Similarly, the MultiJava type system can prevent a method *M* that is declared to be **`final`** from being overridden, in the sense that there can be no other method in the same generic function whose tuple of specializers is a pointwise subtype of *M*'s. Restriction ***R3*** allows this condition to be easily checked. That is, ***R3*** ensures that the methods that a particular method *M* overrides are all available when *M's* compilation unit is typechecked.

5. COMPILATION

We have developed a compilation strategy from MultiJava into standard Java bytecode that retains the modular compilation and efficient single dispatch of existing Java code while supporting the new features of open classes and multiple dispatching. Additional run-time cost for these new features is incurred only where such features are used; code that does not make use of multiple dispatching or external generic functions compiles and runs exactly as in regular Java. MultiJava code can interoperate seamlessly with existing Java code. MultiJava code can invoke regular Java code, including all the standard Java libraries. Additionally, subclasses of regular Java classes can be defined in MultiJava, and regular Java methods can be overridden with multimethods in MultiJava subclasses. Client source code and compiled bytecode is insensitive to whether the invoked method is a regular Java method or a MultiJava multimethod. Aside from the need to import external generic functions, client source code is also insensitive to whether the invoked method is internal or external.

However, internal and external generic functions require different styles of compilation. (Recall that an external generic function is one which has an external top method.) An internal generic function can be compiled as if it were a regular Java method declared inside its receiver class or interface. Internal generic functions are invoked using the same calling sequence as a regular

```
public class Square extends Rectangle {
  /* ... */
  public boolean
    intersect(Shape@Rectangle r) {
      /* method 1 body */
  }
  public boolean
    intersect(Shape@Square s) {
      /* method 2 body */
  }
}
```

Figure 7: Internal generic functions

```
public class Square extends Rectangle {
  /* ... */
  // the "intersect" dispatch method
  public boolean intersect(Shape r) {
    if (r instanceof Square) {
      Square s_ = (Square) r;
      /* method 2 body, substituting s_ for s */
    } else if (r instanceof Rectangle) {
      Rectangle r_ = (Rectangle) r;
      /* method 1 body, substituting r_ for r */
    } else {
      return super.intersect(r);
    }
  }
}
```

Figure 8: Translation of internal generic functions

Java method. An external generic function must be compiled separately from its receiver class or interface. An external generic function uses a different implementation strategy and calling convention than an internal one.

When compiling code that refers to a generic function (either code that adds a method to it or invokes it), the compiler can always tell whether or not the generic function is internal. The compiler has enough information because the code must have imported both the compilation unit declaring the generic function and the one declaring the generic function's receiver type. The generic function is internal if and only if these compilation units are one and the same.

The next subsection describes how declarations and invocations of internal generic functions are compiled. Subsection 5.2 describes the same for external generic functions. Subsection 5.3 describes compilation of super sends. Although compilation is directly to Java bytecode, to simplify discussion we will generally describe compilation as if going to Java source. However, in some situations we need to exploit the additional flexibility of compiling directly to the Java virtual machine.

5.1 Internal Generic Functions

All the multimethods of an internal generic function with the same receiver class are compiled as a unit into a single Java method that we call a *dispatch method*. Consider the set of `intersect` methods in Figure 7. For such a set of multimethods, the MultiJava compiler produces a dispatch method within the receiver class that contains the bodies of all multimethods in the set. Figure 8 shows the result of translating the MultiJava code from Figure 7.[3] In the translation, the dispatch method has the same name as the generic function (`intersect` in this case), and has the same static argument types as all the generic function's methods. The dispatch method internally does the necessary checks on the non-receiver arguments with explicit specializers to select the best of the applicable multimethods from the set. This is implemented using cascaded sequences of **`instanceof`** tests. If multiple paths through these sequences lead to the same method body, `goto` bytecodes could be exploited to avoid the code duplication that would arise in a straightforward compilation to Java source. Alternatively one could compile such a method body into a static method and use a static method call instead of a `goto`. In lieu of cascaded sequences of **`instanceof`** tests, there are other efficient dispatching schemes that could be exploited [Chambers & Chen 99].

3. Of course the compiler must be careful to avoid variable capture.

For the set of multimethods compiled into a dispatch method, the dynamic dispatch tests are ordered to ensure that the most-specific multimethod is found. If one of the multimethods in the set is applicable to some argument tuple, then the typechecking restrictions ensure that there will always be a single most-specific check which succeeds. Moreover, the multimethod body selected by this check will be more specific than any applicable superclass method, so there is no need to check superclass multimethods before dispatching to a local multimethod.

If every multimethod compiled into a dispatch method has an explicit specializer on some argument position, then it is possible that none of the checks will match the run-time arguments. In this case, a final clause passes the dispatch on to the superclass by making a **`super`** call. Eventually a class must be reached that includes a method that does not dispatch on any of its arguments; the modular typechecking rules ensure the existence of such a method when checking completeness of the generic function. In this case, the final clause will be the body of this "default" method.

Compiling regular Java singly dispatched methods is just a special case of these rules. Such a method does not dispatch on any arguments and has no other local multimethods overriding it, and so its body performs no run-time type dispatch on any arguments; it reduces to just the original method body.

An invocation of an internal generic function is compiled just like a regular Java singly dispatched invocation. Clients are insensitive to whether or not the invoked generic function performs any multiple dispatching. The set of arguments on which a method dispatches can be changed without needing to retypecheck or recompile clients.

There is no efficiency penalty for regular Java code compiled with the MultiJava compiler. Only methods that dispatch on multiple arguments get compiled with typecases. A Java program would likely use typecases whenever a MultiJava program would use multimethods anyway, so there should be little performance difference. If a Java program used double-dispatching to simulate multimethods, then it might be possible to generate more efficient code than MultiJava (two constant-time dispatches, plus perhaps some forwarding if inheritance is needed on the second argument), but double-dispatching sacrifices the ability to add new subclasses modularly.

5.2 External Generic Functions

An external generic function must have been introduced by an external method declaration. Since the generic function's receiver class has already been compiled separately, the external generic function cannot be added as a method of that class. Instead, we generate a separate class, called an *anchor class*, to represent the external generic function.

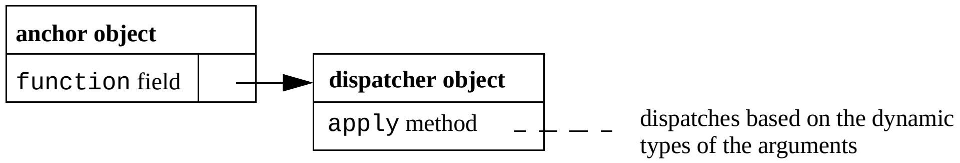


Figure 9: Objects used in the compilation of external generic functions

```
/*  compilation unit "rotate"  */
public Shape Shape.rotate(float a) {
  /*  method 3 body  */
}
public Shape Rectangle.rotate(float a) {
  /*  method 4 body  */
}
public Shape Square.rotate(float a) {
  /*  method 5 body  */
}
```

Figure 10: A compilation unit defining an external generic function

```
public interface rotate$rotate$d {                  //  type of a dispatcher object in this example
  Shape apply(Shape this_, float a);
}

public class rotate$rotate$anchor {                 //  an anchor class

  public static rotate$rotate$d function = new rotate$rotate$dispatcher();

  //  an inner class implementing a dispatcher object
  private class rotate$rotate$dispatcher implements rotate$rotate$d {
    public Shape apply(Shape this_, float a) {
      if (this_ instanceof Square) {
        Square this2_ = (Square) this_;
        /*  method 5 body, substituting this2_  for  this */
      } else if (this_ instanceof Rectangle) {
        Rectangle this2_ = (Rectangle) this_;
        /*  method 4 body, substituting this2_  for  this */
      } else {
        /*  method 3 body, substituting this_  for this  */
      }
    }
  }
}
```

Figure 11: Translation of Figure 10

Figure 9 shows the objects generated in the compilation of external generic functions. An anchor class instance has a single static field, `function`, containing a dispatcher object. During an invocation of the generic function, the dispatcher object is responsible for running one of the generic function's methods based on the dynamic types of the arguments. A dispatcher object is the Java version of a first-class function. It contains all the methods of a particular generic function that are declared in a single compilation unit.

As an example, Figure 10 introduces the `rotate` external generic function and its first three methods. Figure 11 shows the results of compiling it. The privileged access level of the top method determines the privileged access level of the anchor class, its `function` field, and the dispatcher interface. The names for the anchor class and generic function interface are formed by concatenating the name of the compilation unit containing the top method with the generic function name and the appropriate suffix (`anchor` or `d` respectively.) Thus in this example, the anchor class is named `rotate$rotate$anchor` and the dispatcher interface is named `rotate$rotate$d`. As with internal generic functions, dispatching is performed using cascaded **`instanceof`** tests; the same optimizations apply.

To invoke an external generic function, the client loads the dispatcher object from the anchor class's `function` field and invokes its `apply` method on all the arguments to the generic function, including the receiver. So the following MultiJava code:

```
Shape s1 = new Rectangle();
Shape s2 = new Square();
if (s1.intersect(s2)) {
  s2 = s2.rotate(90.0);
}
```

is translated to:

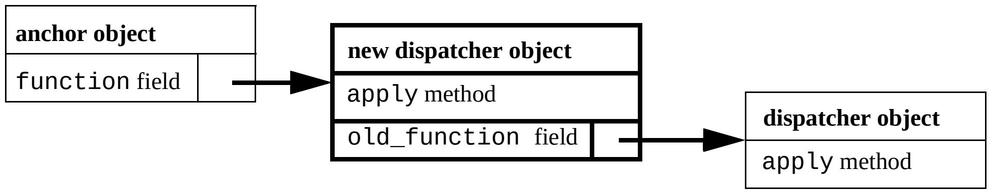


Figure 12: Objects used when adding methods to non-local external generic functions

```
// compilation unit "Oval"
public class Oval extends Shape {
  /* ... */
  public Shape rotate(float a) { /* method 6 body */ }
}
```

Figure 13: Adding a multimethod to an external generic function

```
public class Oval extends Shape {
  // static initializer:
  { rotate$rotate$anchor.function =
      new Oval$rotate$dispatcher(rotate$rotate$anchor.function); }
  /* ... */
  // an inner class implementing a dispatcher object
  private class Oval$rotate$dispatcher implements rotate$rotate$d {
    public rotate$rotate$d oldFunction;
    public Oval$rotate$dispatcher(rotate$rotate$d oldF) { oldFunction = oldF; }
    public Shape apply(Shape this_, float a) {
      if (this_ instanceof Oval) {
        Oval this2_ = (Oval) this_;
        /* method 6 body, substituting this2_ for this */
      } else {
        return oldFunction.apply(this_, a);
      }
    }
  }
}
```

Figure 14: Translation of Figure 13

```
Shape s1 = new Rectangle();
Shape s2 = new Square();
if (s1.intersect(s2)) {
  s2 = rotate$rotate$anchor
        .function.apply(s2,90.0);
}
```

As with internal generic functions, clients invoking external generic functions are insensitive to whether or not the generic function performs any multiple dispatching. Once again the set of arguments on which a method dispatches can be changed without needing to retypecheck or recompile clients.

Next we consider the compilation of methods that add to a non-local external generic function. These additional methods are defined in the same compilation unit as their receiver classes, as required by typechecking restriction ***R3***. There could be several such receiver classes in the same compilation unit. For each of these receiver classes, the translation creates a new dispatcher object to contain the set of the generic function's methods with that receiver class.

Figure 12 shows a new dispatcher object created for such a set of methods. The anchor class's `function` field from Figure 9 is updated to reference this new dispatcher object. In turn, the new dispatcher object contains an `old_function` field that references the original dispatcher object. When the generic function is invoked, the `apply` method of the new dispatcher object is called. It checks if any of its methods are applicable. If none are, it calls the `apply` method of the original dispatcher object (using the `old_function` field).

For example, Figure 13 shows a class, `Oval`, containing a method that is added to the non-local external generic function, `rotate`. Figure 14 shows the results of compiling this class. A new dispatcher class, `Oval$rotate$dispatcher`, is defined whose `apply` method checks whether the run-time arguments should dispatch to the local `rotate` method. The static class initialization for `Oval` creates an instance of this dispatcher object and sets the dispatcher's `old_function` field to the previous dispatcher object (using the dispatcher's constructor). Next the new dispatcher object is assigned to the `function` field.

When invoked, the dispatcher object checks whether the receiver object is an `Oval`. If so, then `Oval`'s `rotate` method is run. If not, then dispatching continues by invoking the `apply` method of the previous dispatcher object (as in the Chain of Responsibility pattern [Gamma *et al.* 95, pp. 223-232]). This may be from some other class that also added methods to the `rotate` generic function. Eventually dispatching either finds a function with an applicable method that was added to the chain, or the search ends at the initial dispatcher object installed when the generic function was created. Completeness checking ensures that this last dispatcher object includes a default method that handles all arguments, guaranteeing that dispatching terminates successfully. While potentially slow, this Chain of Responsibility pattern is only used for compiling external generic functions, which cannot be written in standard Java. There is no efficiency penalty for methods that can be written in standard Java.[4]

The order in which dispatcher objects are checked depends on the order in which they are put into the chain referenced by `rotate$rotate$anchor`'s `function` field. Java ensures that superclasses are initialized before subclasses [Gosling *et al*. 00, §12.4], so dispatcher objects for superclasses will be put onto the chain earlier than subclass dispatchers, causing subclass dispatchers to be checked before superclass dispatchers, as desired. Unrelated classes can have their dispatchers put onto the chain in either order, but this is fine because modular typechecking has ensured that the multimethods of such unrelated classes are applicable to disjoint sets of legal argument tuples, so at most one class's multimethods could apply to a given invocation.

As noted in Subsection 2.2, internal methods that are part of external generic functions are granted access to the private data of their receiver class. To achieve this, the dispatcher object for these methods is compiled as an inner class nested in the corresponding receiver class [Gosling *et al*. 00, §6.6.2].

5.3 Super Sends

The compilation of super sends divides into two cases, depending on whether the super send invokes a method that will be compiled into a different Java (bytecode) method. If the target of the super send will be compiled into a different Java method, then the super send is compiled just as in regular Java, eventually leading to an `invokespecial` bytecode. This implementation strategy is even applicable in the case of a super send from within the dispatcher object of an external generic function, where the class of the dispatcher object is not a subclass of the class referred to by **`super`**. The `invokespecial` bytecode can still be used because this bytecode does not require an inheritance relationship between the caller and callee.

The other possibility is that the super send invokes a method that will be compiled into the same Java (bytecode) method. In this case, the super send should have the effect of one branch of the compiled method invoking a different branch. To avoid duplicating the code of the called branch, we compile such an intra-method invocation into a `jsr` bytecode along with suitable argument and result shuffling code. The callee part should then end with a `ret` bytecode and be invoked via `jsr` from all points in the compiled method where it is reached (both by normal dispatching and by super sends).

6. AN ALTERNATIVE DESIGN

An early plan for adding multimethods to Java was to apply the concept of multiple dispatch as dispatch on tuples [Leavens & Millstein 98], leading to TupleJava. In TupleJava, all multimethods would be external to classes. A multimethod that dispatched at two `Shape` arguments and took an additional non-dispatched `Shape` argument would be declared like

```
public boolean
        (Shape q, Shape r).nearest(Shape s)
  { /* ... */ }
```

and invoked like

```
(myShape1, myShape2).nearest(myShape3)
```

Conceptually invocation is like sending a message to a tuple of objects. TupleJava offers several advantages. The syntax of both defining and invoking a method cleanly separates the dispatched arguments (which occur in the tuple) from the non-dispatched ones (which occur following the method identifier). This separation of arguments maintains a clear parallel between the syntax and the semantics. The tuple syntax also clearly differentiates code that takes advantage of multiple dispatch from standard Java code, which might ease the programmer's transition from a single-dispatch to a multiple-dispatch mind-set.

However, the separation of arguments into dispatched and non-dispatched sets also brings several problems. TupleJava does not provide for robust client code. For example, suppose one wanted to modify the example above to include the dynamic type of the third argument in dispatching decisions. The tuple method declaration above would be rewritten as

```
public boolean
        (Shape q, Shape r, Shape s).nearest()
  { /* ... */ }
```

Furthermore, all method invocations in client code would need to be changed to move the third argument into the tuple. Thus the invocation above would become

```
(myShape1, myShape2, myShape3).nearest()
```

With MultiJava, such a modification requires editing the original method, but all client source code and compiled code can remain unchanged, as such code is insensitive to the set of arguments dispatched upon by the methods of a generic function.

TupleJava also requires all multimethods of a given generic function to dispatch on the same arguments. In particular, this means that multimethods cannot be added to existing singly dispatched methods, which includes all existing Java code. MultiJava does not have this restriction. For example, in MultiJava one could override the `equals` method of the `Object` class to use multiple dispatch as in the following:

```
public class Set extends Object {
  /* ... */
  public boolean equals(Object@Set s)
    { /* ... */ }
}
```

With TupleJava the best one could do is the following:

```
public boolean (Set, Set).equals()
  { /* ... */ }
```

But this attempt would create a new `equals` generic function, completely distinct from the one for testing equality of `Object`s. Thus, with TupleJava, the invocation in the code

```
Object obj1, obj2;
/* ... */
... obj1.equals(obj2) ...
```

will never invoke the special equality operation for `Set`s, even if both arguments have dynamic type `Set`.

4. One can imagine a strategy in which the static initializers that currently add new multimethods to the Chain of Responsibility instead use reflection to analyze the current generic function and then use dynamic compilation to create a new global dispatching method "on-the-fly". The load-time cost of this strategy might be high, but run-time invocation costs could be greatly reduced.

A final argument in MultiJava's favor is that it is strictly more expressive than TupleJava. Indeed, tuple-based method declarations and invocations could be added as syntactic sugar in MultiJava, but not vice-versa.

It remains to be seen whether the ease-of-learning advantages of TupleJava outweigh the expressiveness and code maintenance advantages of MultiJava. We plan to investigate this further once we have completed the implementation of MultiJava.

7. RELATED WORK

The typechecking restrictions for MultiJava are derived from previous work by two of us [Millstein & Chambers 99]. That work presents Dubious, a simple core language based on multimethods and open classes, and describes several type systems for Dubious that all achieve safe static typechecking with some degree of modularity. The type systems differ in their trade-offs between expressiveness, modularity of typechecking, and complexity. We base our MultiJava type system on the simplest and most modular of those systems, called System M.

Encapsulated multimethods [Castagna 95, Bruce *et al.* 95] are a design for adding asymmetric multimethods to an existing singly dispatched object-oriented language. Encapsulated multimethods involve two levels of dispatch. The first level is just like regular single dispatch to the class of the receiver object. The second level of dispatch is performed within this class to find the best multimethod applicable to the dynamic classes of the remaining arguments. The encapsulated style can lead to duplication of code, since multimethods in a class cannot be inherited for use by subclasses. Our compilation strategy for internal generic functions yields compiled code similar to what would arise from encapsulated multimethods, but we hide the asymmetry of dispatch from programmers.

Boyland and Castagna demonstrated the addition of asymmetric multimethods to Java using "parasitic methods" [Boyland & Castagna 97]. To avoid the then-unsolved modularity problems with symmetric multimethods, their implementation is based on the idea of encapsulated multimethods. Parasitic methods overcome the limitations of encapsulated multimethods by supporting a notion of multimethod inheritance and overriding. Parasitic methods are allowed to specialize on interfaces, causing a potential ambiguity problem due to the form of multiple inheritance supported by interfaces. To retain modularity of typechecking, the dispatching semantics of parasitic methods is complicated by rules based on the textual order of multimethod declarations. Additionally, overriding parasitic methods must be declared as parasites, which in effect adds `@` signs on all arguments, but without a clean ability to resolve the ambiguities that can arise in the presence of Java's static overloading. By contrast, our approach offers purely symmetric dispatching semantics and smooth interactions with static overloading, along with modularity of typechecking and compilation. Our approach also supports open classes.

Aspect-oriented programming [Kiczales *et al.* 97] provides an alternative to the traditional class-based structuring of object-oriented programming. Among other things, an aspect may introduce new methods to existing classes without modifying those classes, thus supporting open classes. However, aspects are not typechecked or compiled modularly. Instead, the whole program is preprocessed as a unit to yield a version of the program where the aspects have been inserted into the appropriate classes. Source code is required for all classes extended through aspects, and recompilation of these classes is required if aspects are changed. MultiJava's open class technique does not require the source code for classes that are being extended. Indeed, a client of a library that had no source code access could still add new methods to the classes of that library. MultiJava does not require source code access to the whole program because its static typechecking and compilation are modular. On the other hand, because it cannot edit the whole program's source code and because it does not have pattern-based metaprogramming, MultiJava cannot handle cross-cutting concerns as well as aspect-oriented programming.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how to extend Java with open classes and multimethods. Moreover, we have shown that it is possible to modularly typecheck and efficiently compile these new features. This work extends earlier work on modular typechecking of multimethods [Millstein & Chambers 99] to function properly in a richer programming language (including coping with the existing treatment of single dispatching, static overloading, and compilation units). This extension also supports **`super`** sends and compilation that is efficient, modular, and interoperable with existing Java code.

There are several possible areas for future work. Work continues on the implementation of our MultiJava compiler. Extensions to MultiJava to support top-level static methods, static fields, instance fields, and instance constructors could be investigated. Some straightforward extensions to Java's reflection API could also be considered, for example to answer queries on methods added via the open class mechanism. Finally, further increases in MultiJava's expressiveness could be studied. One area of interest is replacing some of the static typechecking restrictions with static warnings, backed up by link-time checking. Among other things, this change could allow the declaration of abstract external methods and the declaration of top-level methods in arbitrary compilation units, provided their use does not lead to incompleteness or ambiguity at link time.

9. ACKNOWLEDGMENTS

The work of Gary Leavens was supported in part by NSF grant CCR-9803843. Craig Chambers and Todd Millstein were supported in part by NSF grant CCR-9970986, NSF Young Investigator Award CCR-945776, and gifts from Sun Microsystems and IBM. Part of this work was performed while Craig Chambers was on sabbatical at Carnegie Mellon University.

Thanks to Clyde Ruby and Gerald Baumgartner for several discussions about this work and to the anonymous reviewers for their helpful comments.

10. REFERENCES

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. *OOPSLA'91 Conference Proceedings*, Phoenix, AZ, October, 1991, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 113-128. ACM, New York, November, 1991.

[Arnold & Gosling 98] Ken Arnold and James Gosling. *The Java Programming Language*. Second Edition, Addison-Wesley, Reading, Mass., 1998.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 302-315. ACM, New York, January 1997.

[Boyland & Castagna 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 32, number 10 of *ACM SIGPLAN Notices*, pp. 66-76. ACM, New York, November 1997.

[Bruce *et al.* 95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, **1**(3):221-242, 1995.

[Cardelli 88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, **76**(2/3): 138-164, February-March, 1988. An earlier version appeared in *Semantics of Data Types Symposium*, LNCS 173, pp. 51-66, Springer-Verlag, 1984.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, June, 1992, pp. 182-192, volume 5, number 1 of *LISP Pointers*. ACM, New York, January-March, 1992.

[Castagna 95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431-447, 1995.

[Castagna 97] Giuseppe Castagna. *Object-Oriented Programming A Unified Foundation*, Birkhäuser, Boston, 1997.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. *ECOOP '92 Conference Proceedings*, Utrecht, the Netherlands, June/July, 1992, volume 615 of *Lecture Notes in Computer Science*, pp. 33-56. Springer-Verlag, Berlin, 1992.

[Chambers 95] Craig Chambers. The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University of Washington, December, 1995. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html.

[Chambers & Chen 99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. *OOPSLA '99 Conference Proceedings*, pp. 238-255, October, 1999. Published as *SIGPLAN Notices* 34(10), October, 1999.

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, **17**(6):805-843. November, 1995.

[Chambers 98] Craig Chambers. Towards Diesel, a Next-Generation OO Language after Cecil. Invited talk, *The Fifth Workshop on Foundations of Object-oriented Languages*, San Diego, California, January 1998.

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. *Foundations of Object-Oriented Languages*, REX School/Workshop Proceedings, Noordwijkerhout, the Netherlands, May/June, 1990, volume 489 of *Lecture Notes in Computer Science*, pp. 151-178. Springer-Verlag, New York, 1991.

[Feinberg *et al*. 97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[Findler & Flatt 98] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. *International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[Gamma *et al.* 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Gosling *et al*. 00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*, Second Edition. Addison-Wesley, Reading, Mass., 2000.

[Ingalls 86] D. H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, Portland, Oregon, November, 1986, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 347-349. ACM, New York, October, 1986.

[Kiczales *et al.* 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In proceedings of the *Eleventh European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241. June 1997.

[Leavens & Millstein 98] Gary T. Leavens and Todd D. Millstein. Multiple Dispatch as Dispatch on Tuples. *Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, October 1998.

[Martin 98] Robert C. Martin. Acyclic Visitor. In Robert C. Martin, Dirk Riehle, Frank Buschmann (editors), *Pattern Languages of Program Design 3*, pp. 93-103. Addison-Wesley Longman, Inc., Reading, Mass., 1998.

[Millstein & Chambers 99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. In proceedings of the *Fourteenth European Conference on Object-Oriented Programming*, Lisbon, Portugal, July 14-18, 1999. Volume 1628 of Lecture Notes in Computer Science, pp. 279-303, Springer-Verlag, 1999.

[Millstein & Chambers] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. To appear in *Information and Computation*.

[Mugridge *et al.* 91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-Methods in a Statically-Typed Programming Language. *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991, volume 512 of Lecture Notes in Computer Science; Springer-Verlag, New York, 1991.

[Nordberg 98] Martin E. Nordberg III. Default and Extrinsic Visitor. In Robert C. Martin, Dirk Riehle, Frank Buschmann (editors), *Pattern Languages of Program Design 3*, pp. 105-123. Addison-Wesley Longman, Inc., Reading, Mass., 1998.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 146-159. ACM, New York, January 1997.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Shalit 97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language* (*second edition*). Digital Press, Bedford, MA, 1990.

[Vlissides 99] John Vlissides. Visitor into Frameworks. *C++ Report*, **11**(10):40-46. SIGS Publications, November/December 1999.