MultiJava: Modular Symmetric Multiple Dispatch and Extensible Classes for Java

Curtis Clifton  
Iowa State University

Gary T. Leavens  
Iowa State University

Craig Chambers  
Iowa State University

Todd Millstein  
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation
Clifton, Curtis; Leavens, Gary T.; Chambers, Craig; and Millstein, Todd, "MultiJava: Modular Symmetric Multiple Dispatch and Extensible Classes for Java" (2000). Computer Science Technical Reports. Paper 56.
http://lib.dr.iastate.edu/cs_techreports/56
MultiJava: Modular Symmetric Multiple Dispatch and Extensible Classes for Java

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein

TR #00-06
April 2000

Keywords: Multimethods, generic functions, object-oriented programming languages, single dispatch, multiple dispatch, encapsulation, modularity, static typechecking, subtyping, inheritance, open objects, extensible classes, external methods, Java language, MultiJava language, separate compilation.


Submitted for publication.

Copyright © Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein, 2000.
MultiJava: Modular Symmetric Multiple Dispatch and Extensible Classes 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
Multiple dispatch offers several well-known advantages over the single dispatching of conventional object-oriented languages, including a simple solution to the “binary method” problem and cleaner implementations of the “strategy” and similar design patterns. Extensible classes allow one to extend the set of methods that an existing class supports without editing that class or client code. This provides, among other idioms, a simple implementation of the “visitor” design pattern. We present MultiJava, a backward-compatible extension to Java supporting symmetric multiple dispatch and extensible classes. We adapt previous theoretical work to allow MultiJava classes 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 multiple dispatching or extensible classes are actually used.

Keywords
Multimethods, generic functions, object-oriented programming languages, single dispatch, multiple dispatch, encapsulation, modularity, static typechecking, subtyping, inheritance, open objects, extensible classes, external methods, Java language, MultiJava language, separate compilation

1. INTRODUCTION
In object-oriented languages with multimethods (such as Common Lisp, Dylan, and Cecil), the appropriate method to invoke for a message send can depend on the run-time class of any subset of the argument objects. This mechanism is called multiple dispatch, in contrast with single dispatch found in languages like Smalltalk and Java. In single dispatch methods are selected based on just the run-time class of a distinguished receiver argument. Multimethods support safe covariant overriding in the face of subtype polymorphism, providing a natural solution to the “binary method” problem [Bruce et al. 95], and a simple implementation of the “strategy” design pattern [Gamma et al. 95, pp. 315-323].

Extensible classes allow clients to extend the set of methods that an existing class supports without editing that class or any other existing code, allowing a form of what we have called “open objects” [Chambers 98]. Extensible classes enable easy programming of the “visitor” design pattern [Gamma et al. 95, pp 331-344, Baumgartner et al. 96] and are a key element of aspect-oriented programming [Kiczales et al. 97]. With extensible 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.

A major obstacle to adding 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 class 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 & Millstein 98] or has sacrificed the natural symmetric multimethod dispatching semantics in order to ensure modularity [Boylan & Castagna 97].

In this paper we add multiple dispatch and extensible classes to the Java programming language [Gosling et al. 96, Arnold & Gosling 98]; we call our extension MultiJava. We maintain both the natural symmetric multimethod dispatching semantics as well as the static modular typechecking and compilation properties of standard Java. We do this by adapting previous work of two of the authors on modular static typechecking for 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 multiple dispatching and extensible classes 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 retains backward-compatibility and interoperability with existing Java source and bytecode.

In the following section we motivate the need for multiple dispatching and extensible classes. We informally describe the syntax and dynamic semantics of MultiJava in Section 3. In Section 4 we discuss MultiJava’s static type system. In Section 5 we outline the compilation of MultiJava source code into Java bytecode and in Section 6 we discuss some of our design decisions and their motivations. Section 7 discusses related work and Section 8 discusses several avenues for future work.

2. LIMITATIONS OF JAVA
Although successful, Java is not able to easily express several common programming idioms. Among these are “binary methods”
and the addition of new methods to existing class hierarchies.

2.1 Binary Methods

Binary methods are methods that operate on two or more objects of the same class [Bruce et al. 95]. Such methods arise in common programming situations. As an example, consider the Shape class below, whose method for checking whether two shapes intersect is a binary method.

```java
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:

```java
public class Rectangle extends Shape {
    /* ... */
    public boolean intersect(Rectangle r) {
        /* ... */
    }
}
```

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 type-checking 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 `sl.intersect(s2)` in Java would invoke the overriding method whenever `sl` 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.intersect`'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.intersect`'s method to `statically overload` `Shape.intersect`'s method. Conceptually, one can think of each method in the program as implicitly belonging to a `generic function`, a collection of methods consisting of a `top method` and all of the methods that (dynamically) override it. In particular, given a method `M_sub` in class `C`, if there is a method `M_sup` of the same name, number of arguments, and argument types as `M_sub` in some superclass or superinterface of `C`, 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.

In our example, the two `intersect` methods belong to different generic functions. Java uses the name, number of arguments, and `static argument types` of a message send to statically determine which generic function it invokes. So in particular, Java determines statically which of the two methods will be invoked for each intersection message send, just as if the methods had two different names. For example, consider the following client code:

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

Although the two `intersect` message sends above have identical parameters, they invoke different methods. In particular, the second message send will invoke the `Shape.intersect` method, even though both `s1` and `s2` are `Rectangle` instances, because of the static types of these arguments. This means that subtype polymorphism on the `Shape` hierarchy is lost for `intersect`: existing client code of the `Shape` class that calls `intersect` will never invoke the more efficient intersection method, even if the arguments are both `Rectangles` at run-time.

In Java, one can solve this problem by performing explicit run-time type tests and associated casts. For example, one could implement the `Rectangle.intersect` method as follows:

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

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

A related solution to the binary method problem in Java is the use of double dispatching [Ingalls 86]. 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`, and the name of the message encodes the dynamic class of the original receiver. Like the type-test technique, programming double dispatching by hand is very tedious and error-prone. Double dispatching also requires existing classes to be modified to add new methods whenever new subclasses (like `Triangle`) are added. An example of this technique is employed in the visitor design pattern example discussed in the next subsection. This technique is also the essence of the “strategy” design pattern.

2.2 Adding New Methods to Existing Classes

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
Instances 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 re-typecheck and re-compile 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 interface. However, leaving the operation out of the class’s interface 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. Although 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, 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 class hierarchy in Figure 1. Instances of these classes are used by a compiler to form abstract syntax trees. The compiler might perform several operations on these abstract syntax trees, such as typechecking and code generation. Visitor classes are defined for each of these operations. We give Java code for the classes WhileLoopNode, IfThenNode, and TypeCheckingVisitor in Figure 2. The visit method of each kind of node uses double dispatching to invoke the method appropriate for that type of node. In this way, a new operation can be added modularly; the programmer simply introduces a new Visitor subclass containing methods for visiting each class in the Node hierarchy.

Unfortunately, this approach has several disadvantages. First, the code suffers from the drawbacks of double dispatching mentioned in the previous section. Second, the visitor pattern must be planned for ahead of time. For example, had the Node hierarchy not been written with a visit method, which allows visits from the Visitor hierarchy, it would not have been possible to add typechecking functionality in a modular way. Even with the visit 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 visitor subclass instance. Finally, although this technique 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, all of the Visitor subclasses would have to be modified to contain a method for visiting the new kind of node.

3. MULTIJAVA

This section informally describes the syntax and semantics of MultiJava and provides solutions to the problems outlined in the previous section. Our extension to the Java syntax appears in Figure 3. In the figure we enclose optional components in square brackets (’[’ and ’]’) and use an asterisk (*) to denote zero or more occurrences of a component.

In the next subsection we discuss the syntax and semantics of multimethod declarations, including interaction with static overloading. In Section 3.2 we describe MultiJava’s extensible class feature and its implications on several aspects of standard Java. Section 3.3 describes upcasting, a generalization for multimethods of Java’s super construct.

3.1 Multimethod Declarations

MultiJava allows programmers to write multimethods, methods that can dynamically dispatch on arguments other than the receiver object. We illustrate MultiJava’s multimethod syntax by providing a solution to the binary method example of Section 2.1. The Shape class is defined as in that example, and the Rectangle class is defined as follows:

public class Rectangle extends Shape {
    /* ... */
    public boolean intersect(Shape@Rectangle r) {
        /* ... */
    }
}

This code is identical to the first solution attempt presented in Section 2.1, except for the type declaration of the formal parameter r, which is Shape@Rectangle instead of simply Rectangle. The Shape part denotes the static type of the argument r. In particular, this tells us that the method belongs to the same generic function as the Shape class’s intersect method, because the name, number of arguments, and (static) argument types match. The @Rectangle indicates that we wish to dynamically dispatch on this argument. In particular, this method requires the dynamic class of the argument r to be a subclass of Rectangle. Rectangle is referred to as the explicit specializer of argument r, and thus this argument position is specialized at Rectangle. The typechecking rules require that explicit specializers are classes.

For a given multimethod M, its tuple of specializers (S₀,...,Sₙ) is such that S₀ is M’s receiver type and, for i ∈ {1,...,n}, if M has an explicit specializer, Uᵢ, at the iᵗʰ position, then Sᵢ is Uᵢ, otherwise Sᵢ is the static type of the iᵗʰ argument.

Multimethods in MultiJava use symmetric dispatching semantics, a natural generalization of Java’s dynamic dispatching semantics. For a message send Eᵢp.R(Eᵢ,...Eᵢp), we evaluate each Eᵢ to some value vᵢ, extract the methods that constitute the generic function being invoked, and then select and invoke the most-specific applicable such method for the arguments (vᵢ,...,vᵢp). Let (C₀,...,Cₙ) be the
Asymmetric multiple dispatch typically uses lexicographic ordering, where earlier arguments are more important for resolving dispatching. A third option uses textual ordering of methods to determine which applicable method is most specific. We believe that the symmetric multiple dispatch semantics is the most intuitive and least error-prone, reporting possible ambiguities rather than silently resolving them in potentially unexpected ways.

When declaring a multimethod one can often omit the static type of some formal parameters. This can be done when the generic function the multimethod belongs. If a multimethod could belong to more than one generic function, based on its name and whether it requires the use of the static type parts. For example, suppose that the Shape class contained a second method for intersect, where Quadrilateral is an interface that Rectangle implements:

```java
public boolean intersect(@Rectangle r) { /* ... */
```
public boolean intersect(Quadrilateral q) {
    /* ... */
}

By Java’s static overloading rules, this is a top method that defines a
different generic function than the one that takes Shape as an
argument. In this case, the following declaration would be
ambiguous:

public boolean intersect(Rectangle r) {
    /* ... */
}

and MultiJava requires the full specification Shape@Rectangle
or Quadrilateral@Rectangle to determine to which of the
two intersect generic functions the new method belongs.

3.2 Extensible Classes
The extensible class feature of MultiJava allows a programmer to
easily add new methods to existing classes without modifying
existing code. Consider again the compiler example of Section 2.2.
To add typechecking functionality to the existing Node hierarchy,
MultiJava uses extend class declarations to hold the new
methods. For example, the functionality of the typechecking visitor
from Figure 2 can be written as follows:

package oopsla.examples;

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

A program may contain several extend class declarations that
extend the same class. As in Java, the bodies of methods in an
extend class declaration may use this to reference the
receiver object.

Clients may use the new methods exactly as they would use the
class’s original methods. For example, typechecking on
rootNode is performed as follows:

rootNode.typeCheck()

where rootNode is an instance of Node or a subclass.

Extensible classes completely obviate the need for the visitor
pattern infrastructure; the Visitor class hierarchy and visit
methods of the Nodes are now unnecessary. Instead the operations
to be performed are written as extensions of the Node subclasses,
effectively creating a new generic function. We call such methods
external methods, and the generic function an external generic
function. Unlike with the visitor pattern, there is no need to plan
ahead for adding the new generic functions. Additionally, each
family of external methods (i.e., each generic function) can define
its own argument types and result type, independently of other
visitors. 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. Regular method
inheritance allows all new subclasses to inherit the methods defined
in the external generic function.

An extend class declaration need not be written in the same
compilation unit as the associated class declaration. (A Java
compilation unit corresponds to a single file in Java
implementations based on file systems [Gosling et al. 96, §7.6].)
For example, the above typeCheck methods can all be put in
a single file separate from the compilation units defining the classes
of the Node hierarchy. This allows 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.

To invoke or extend 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. Once the generic
function is made visible, it can be invoked. Additionally, new
methods can be added to the generic function. For example, a
compilation unit defining a new subclass of Node can import the
typeCheck compilation unit and then define a typeCheck
method on the new Node subclass.

By making import explicit, client code can manage the name spaces
of the classes they manipulate. Only clients that explicitly 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.

Java allows at most one public type (class or interface)
declaration in a compilation unit [Gosling et al. 96, §7.6]. 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, or the methods of one public generic function that
extends a type from a different compilation unit. A single
compilation unit can contain both a public class and several
external generic functions and methods on that class. We allow this
because the methods in these external declarations could just as
easily have been written in the original class declaration.

An important issue is what kind of privileged access to grant to
external methods. Since we desire MultiJava to retain the same
encapsulation properties as Java [Gosling et al. 96, §6.6], we allow
an external method to access:

• public members of the class it extends,
• non-private members of the class it extends if the external
  method is in the same package as that class, and
• all members of the class it extends if the external method is in
  the same compilation unit as that class.

All other access is prohibited. In particular, an external method
does not in general have access to the private data of the receiver.

Several method definitions may occur within each extend class
declaration, just as in an ordinary class declaration. However, a
common idiom, as in our example above, is to introduce a single
new generic function along with several new methods. For cases
like this we provide a syntactic sugar that allows for the top-level
description of methods. For example, the new typechecking
methods given at the beginning of Section 3.2 can equivalently be
written as follows:

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.
package oopsla.examples;

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

3.3 Upcasting

To support a notion of **super** calls for multimethods, MultiJava includes **upcasting**. For example, to invoke the default **intersect** method in the Shape class on two Rectangles, we can write the following:

```java
(@Shape)r1.intersect((@Shape)r2)
```

The syntax is similar to a regular Java cast, except for the presence of the at-sign (@). When the above message is sent, the upcasts force the call to be dispatched as if the run-time types of both r1 and r2 were Shape, thereby invoking the Shape class's **intersect** method. (In this particular example, only one of the upcasts is necessary to obtain the desired effect.) If the @ signs were removed, resulting in regular Java casts, the message send would instead invoke the Rectangle class's **intersect** method. Java's regular super send, where the receiver of a message is **super**, becomes syntactic sugar for a MultiJava upcast with the receiver (**Superclass**). This, where Superclass is the name of the immediate superclass of the receiver's static type.

4. TYPECHECKING

In this section we describe how to extend Java's static type system to accommodate MultiJava's extensions. We discuss the new type system in Section 4.1. A key goal of MultiJava is to maintain the static modular typechecking approach of standard Java. Section 4.2 discusses several challenges that multimethods and external methods pose for a modular typechecking approach. In Section 4.3 we discuss the restrictions imposed to achieve modularity and show how to resolve the problems described in the previous subsection. Section 4.4 describes static typechecking of our upcast mechanism.

4.1 Static Typechecking

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 and implementation-side checking of message sends and methods [Chambers & Leavens 95].

**Client-side checks** are local checks for type correctness of each message send expression. For each message send expression \( E_0.R.E_1,...,E_n \) in the program, let \( T_i \) be the static type of \( E_i \). Then there must exist a unique generic function named \( f \) 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.

**Implementation-side checks** ensure that each generic function is fully and unambiguously implemented. These checks have two parts. First, the following checks are performed on each method declaration in isolation:

- Its privileged access modifiers must match those of its top method.
- For each of its explicit specializers, \( S \), the associated static type must be a supertype of \( S \) and \( S \) must be a class.

Requiring the explicit specializers to be classes eliminates an ambiguity that could arise if two methods in the same generic function specialized on incomparable interfaces, since interfaces support a kind of multiple inheritance.

A second part of the implementation-side checks tests all the multimethods in a generic function as a group. Consider a generic function with name \( f \) whose top method has argument types \( (T_0,...,T_n) \). We say that a tuple of classes \( (C_0,...,C_n) \) 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. The checks are that for each generic function, there is a most-specific applicable method for each of its legal argument tuples. For example, consider implementation-side checks on the **intersect** generic function, defined by the Shape class in Section 2.1 and the Rectangle class in Section 3.1. There are four 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 legal argument tuples, but more efficient algorithms exist that only check the “interesting” subset of tuples [Chambers & Leavens 95, Castagna 97].

The client-side checks and the first part of the implementation-side checks are inherently modular, and performed on a single message send expression or method in the program. However, the second part of the implementation-side checks require complete knowledge of a generic function’s methods and its legal argument tuples. Therefore, the previous attempt to add symmetric multimethods to existing object-oriented languages required global implementation-side typechecking [Leavens & Millstein 98].

In contrast, we propose a modular implementation-side typechecking approach. In particular, each compilation unit is required to check only new legal argument tuples of generic functions that are imported. This is a generalization of Java’s current modular checks. Consider again implementation-side typechecking for the **intersect** generic function defined by the Shape class from Section 2.1 and the Rectangle class from Section 3.1. Assume the two classes are created in different compilation units. When the Shape class is created, its compilation unit is checked to see that the (Shape,Shape) argument tuple has a most-specific intersect method, as that is the only legal argument tuple for the **intersect** generic function. When the Rectangle class is created, along with a new multimethod for **intersect**, MultiJava checks the other three legal argument tuples to **intersect**, but it need not re-check the (Shape,Shape) tuple.

4.2 Challenges for Modular Typechecking

Unfortunately, the modular implementation-side typechecking approach described above is unsound without further restrictions. In previous work [Millstein & Chambers 99], we described scenarios where modular typechecking can miss message-not-understood or message-ambiguous errors that may occur at run-time. We review these scenarios here, adapting them to the Java context.

4.2.1 Unrestricted Multimethods

Message-ambiguous errors that elude static detection can be caused by unrestricted multimethods. Consider the example in Figure 4. Suppose the **extend class** declaration and the class in the example are in compilation units \( U_i \) and \( U_j \), respectively. Unit \( U_i \) must import both the Shape and Rectangle classes, while \( U_j \) imports only the Shape class. The declaration that extends the class Shape in \( U_i \) adds a method that overrides the default Shape **intersect** method for arguments whose dynamic class is
Abstract classes or interfaces coupled with multimethods can lead to message-not-understood errors. Consider the example in Figure 4, assuming each class is in its own compilation unit. Since the Picture class is declared abstract, it need not implement the similar method. The JPEG class’s compilation unit checks that the single legal argument tuple in scope, (JPEG, JPEG), has a most-specific similar method, and similarly for the GIF class’s 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. We say a generic function is incomplete if it can cause message-not-understood errors when invoked.

4.2.3 Extensible Abstract Classes or Interfaces
Extensible abstract classes or interfaces also can lead to message-not-understood errors. Consider the example in Figure 6, assuming each top-level declaration is in its own compilation unit. The JPEG class is a concrete implementation of the abstract Picture class, providing an appropriate similar method. However, the unseen extend class declaration adds a new abstract method, draw, to the Picture class, which is allowed because Picture is abstract. However, if a client ever invokes draw on a JPEG, a message-not-understood error will occur.

4.3 Restrictions
In this section, we adapt some of our earlier work [Millstein & Chambers 99], which provides safe and modular static typechecking of multimethods and open objects, to Java. That work contains several alternative type systems that have been proved to be sound; here we use the most modular, called System M.

To explain the adaptation of this system to MultiJava we start with a few definitions. 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. A class is concrete if it is not declared abstract. A non-concrete type is either an abstract class or an interface. We say that a type $T_1$ is a descendant of type $T_2$ if $T_2$ is a (possibly reflexive or transitive) supertype of $T_1$. A type is visible in a compilation unit $U$ if it is declared in or referred to in $U$. 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 $U$. Two compilation units are unrelated if neither imports the other.

To avoid the multimethod ambiguity problem of Section 4.2.1, MultiJava ensures that, for a given generic function, multimethods declared in unrelated compilation units are applicable to disjoint argument tuples. Restriction $R1$ ensures that the specializers of such methods differ in the first argument position:

**(R1)** If a method $M_j$ declared in an extend class declaration for a class $C$ overrides a non-local method $M_z$, then $C$ must be a local class.

(Absence of ambiguity is ensured for multimethods declared in a single compilation unit by the implementation-side checks described in Section 4.1.)

To avoid the incompleteness scenario of Section 4.2.2, restriction $R2$ ensures that a multimethod that specializes on a non-concrete type provides a default implementation for that type:

**(R2)** If a generic function has the static type $T$ at some argument position other than the receiver, then implementation-side typechecks of the generic function must consider all non-concrete visible descendants of $T$ to be concrete at that argument position.

Restriction $R3$ provides similar guarantees for the situation of Section 4.2.3:

**(R3)** If a local generic function’s top method has the non-local
type $T$ as the receiver, then implementation-side typechecks of the generic function must consider any non-local, non-concrete visible descendants of $T$ to be concrete at the receiver position.

In essence, the top method of an external generic function cannot be abstract. However, local generic functions with abstract receiver classes can be abstract, leaving concrete subclasses to provide appropriate concrete method implementations.

These restrictions resolve the problems in the examples of the previous subsection. In Figure 4, the declaration that extends the class $Shape$ violates restriction $R1$. In particular, the associated intersect method overrides the non-local intersect method in the $Shape$ class, but the $Shape$ class declaration is not local to the current compilation unit. On the other hand, if the $Shape$ class declaration were local, the restriction would be satisfied. In that case, the $extend$ class declaration would be visible to the $Triangle$ class's compilation unit, which would therefore statically detect the ambiguity. Restriction $R1$ is similar to the implicit restriction imposed by the syntax of encapsulated multimethods [Castagna 95, Bruce et al. 95], but kept entirely in the static type system; the symmetric semantics of multimethod dispatching is not changed.

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 Figure 6, the $draw$ method in the $extend$ class declaration for $Picture$ introduces a new generic function with a non-local, non-concrete receiver. By restriction $R3$, implementation-side typechecks must consider $Picture$ to be concrete, thereby finding an incompleteness for the legal argument tuple (Picture). Therefore, the $extend$ class declaration must provide an implementation for drawing $Picture$, which resolves the incompleteness for the unseen $JPEG$ class.

In summary, our static type system retains modularity of typechecking in Java, while safely providing symmetric multimethods and extensible classes. Multimethods with a non-local receiver that override a non-local multimethod are disallowed, but all other multimethod idioms are safe, including the common case of binary methods [Millstein & Chambers 99]. In addition, arbitrary external operations may be added to existing classes modularly, as long as the appropriate default method implementations are included to prevent unseen incompleteness.\(^2\)

### 4.4 Typechecking Upcasts

Upcast message sends require several checks in addition to the ordinary client-side checks for message sends. Suppose the actual argument at position $i$ is upcast to type $T$. Then $T$ must be a supertype of the static type of the $i$th actual argument, ensuring that the upcast will succeed at run-time. Further, $T$ must be a subtype of the $i$th argument type of the associated generic function's top method, ensuring that an instance of type $T$ can safely be passed to the generic function at the $i$th argument position.

If all of the upcasts are to concrete classes, then the above checks are sufficient to ensure type safety of the message send. However, if at least one of the upcasts is to a non-concrete type, then it is possible that no most-specific applicable method exists for the message send. Consider the following example:

```java
public abstract class Picture {
    public abstract boolean similar(Picture p);
}
public class JPEG extends Picture {
    public boolean similar(Picture p) {
        @Picture this.similar(p);
    }
}
```

The upcast message send from JPEG's similar method satisfies the restrictions described at the beginning of this subsection. However, the message send will cause a message-not-understood error at run-time, because there is no similar method implementation in the $Picture$ class.

Therefore, in this case we must additionally check that the upcast message send will indeed have a most-specific applicable method to invoke. In our example, no visible method covers the ($Picture$, $Picture$) argument tuple, resulting in a static type error. This check is a generalization of the check that Java already needs to safely allow super calls to non-concrete types. For example, the above upcast example is equivalently written as

```java
super.similar(p);
```

In this case, Java checks for a similar implementation in the $Picture$ class and signals a static error because such a method does not exist.

### 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 multiple dispatching and extensible classes. 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 generic functions can be overridden with multimethods in MultiJava subclasses. Clients of such generic functions, either in regular Java or MultiJava, are insensitive to whether the invoked method is a regular Java method or a MultiJava multimethod.

There are two styles of implementation of a generic function, depending on whether the generic function's top method and the method's receiver type were declared in the same compilation unit or not. A top method with a local receiver type can be compiled as if it were a regular Java method declared inside its receiver class or interface, whether it was written inside the class or via an $extend$ class declaration of a local class. When a top method has a local receiver we call the resulting generic function and its methods $encapsulatable$. Encapsulatable generic functions are invoked using the same calling sequence as a regular Java method. A generic function whose top method has a non-local receiver type, which necessarily was written in an $extend$ class declaration, must be compiled separately from its receiver class or interface. Such a non-$encapsulatable$ generic function uses a different implementation strategy and calling convention than encapsulatable generic functions.

\(^2\) Rather than rejecting programs that fail our restrictions, we could provide only warnings. A later link-time check could then recheck such potentially incorrect generic functions to see whether the potential error occurred in the particular program being linked. In this way, we could allow MultiJava programs that do not combine classes in unsafe ways while still providing a modular warning that some combinations could be unsafe.
A client of a generic function, either one that adds a new method or one that invokes it, can always tell whether or not the generic function is encapsulatable based solely on information available at compile time. Such a client must have imported (directly or indirectly) both the compilation unit declaring the generic function and the one declaring the generic function’s receiver type. The generic function is encapsulatable if and only if these compilation units are one and the same.

The next subsection describes how declarations and invocations of encapsulatable generic functions are compiled. Subsection 5.2 describes the same for non-encapsulatable generic functions. Subsection 5.3 describes compilation of upcasts. For the most part, we will describe compilation as if going to Java source. In some situations we will exploit the additional flexibility of compiling directly to the Java virtual machine.

5.1 Encapsulatable Generic Functions

All the multimethods of an encapsulatable generic function with the same receiver class are compiled as a unit into a single Java method. Consider the set of intersect methods in Figure 7. For such a set of multimethods, the MultiJava compiler produces a method within the receiver class that contains the bodies of all multimethods in the set. The generated 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. Figure 8 shows the result of translating the MultiJava code from Figure 7.3 Methods that dispatch on multiple arguments lead to cascaded sequences of instanceof tests. If multiple paths through these sequences lead to the same method body, goto bytecodes can be exploited to avoid the code duplication that would arise in a straightforward compilation to Java source. Other efficient dispatching schemes for such cascaded sequences of instanceof tests have been developed which could be exploited in an implementation of MultiJava [Chambers & Chen 99].

When one method overrides another within the same set of methods being compiled, the dynamic dispatch tests are ordered such that the most specific argument tuples are checked before ones that they override. If one of the multimethods in the set is applicable to the argument types 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 all the local multimethods dispatch on arguments, then it is possible than none of the cases 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 encapsulatable 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 retypcheck or recompile clients.

5.2 Non-Encapsulatable Generic Functions

A non-encapsulatable generic function must have been introduced as part of an extend class declaration of a non-local class. Since the non-local class has already been compiled separately, the non-encapsulatable generic function cannot be added as a method of that class. Instead, we generate a separate class, called the anchor class, to represent the non-encapsulatable generic function. The anchor class has a single static field, function, containing an object with an apply method (the Java version of a first-class function object) that is invoked to perform the necessary dispatching of the generic function. The function field is initialized to an object whose apply method dispatches to one of the multimethods declared in the same compilation unit as the generic function (i.e., as its top method). Another compilation unit that adds methods to the non-encapsulatable generic function updates its function field with a different object whose apply method checks for dispatching to its multimethods, invoking the previous contents of the function field if none of its multimethods apply. The modular typechecking restrictions, along with Java’s initialization of superclasses before subclasses, ensure that more specific multimethods are checked for before less specific ones, guaranteeing the proper dispatching semantics.

As an example, Figure 9 introduces a non-encapsulatable generic function and its first three methods. Figure 10 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 generic function 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 qf respectively.) Dispatching is performed using cascaded instanceof tests, just as dispatching over arguments is performed for encapsulatable multimethods.

3. Of course the compiler must be careful to avoid variable capture.
public Shape Shape.rotate(float a) {
    /* method 3 body */
}

public Shape Triangle.rotate(float a) {
    /* method 5 body */
}

Figure 9: Non-encapsulatable multimethods.

public interface Rotate_rotate_gf {
    Shape apply(Shape this_, float a);
}

Figure 10: Translation of non-encapsulatable multimethods.

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

Figure 11: Multimethod extending a non-encapsulatable generic function.

Figure 12: Translation of a multimethod extending a non-encapsulatable generic function.
To invoke a non-encapsulatable generic function, the client loads the 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:

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

is translated to:

```java
Shape s1 = new Rectangle();
Shape s2 = new Triangle();
if (s1.intersect(s2)) {
    s2 = Rotate_rotate_anchor .function.apply(s2,90.0);
}
```

To compile a set of multimethods that extend a non-local non-encapsulatable generic function, a function object is created to handle the dispatching to these additional methods. These extending multimethods are defined in the same compilation unit as their receiver classes, as required by typechecking restriction RI. Such multimethods are granted access to private data of their receiver class. Consequently, a separate function object performing the dispatching is compiled for each distinct multimethod receiver class, and this function object is an inner class nested in the corresponding receiver class. Static initialization code for each receiver class adds the new dispatching code to the chain of dispatchers.

Figure 11 presents an example class, Oval, containing a method that extends a non-local non-encapsulatable generic function, rotate. Figure 12 shows the results of compiling this class; this translation would be unchanged if rotate were defined in an extend class declaration in the same compilation unit. A new dispatching class, Rotate_rotate_genericFunction, is defined whose apply method checks whether the run-time arguments should dispatch to the local rotate method. The static class initialization for Oval will create an instance of this dispatching object, remember the previous dispatching object stored in the function field of the Rotate_rotate_anchor class, and update the function field to hold the new dispatching object. When invoked, the dispatching object will check 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 dispatching object for Rotate_rotate (as in the Chain of Responsibility pattern [Gamma et al. 95, pp. 223-232]). This may be from some other class that also extended the rotate generic function. Eventually dispatching either finds an applicable method from a class that extended the generic function, or the initial dispatching function object installed when the generic function was created is invoked. Completeness checking ensures that this last dispatching function includes a default method that handles all arguments, guaranteeing that dispatching terminates successfully.

The order in which dispatch 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. 96, §12.4], so dispatching 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 have no common argument tuples, so at most one class's multimethods could apply to a given invocation.

### 5.3 Upcasts

The compilation strategy described above is not sufficient for handling MultiJava's upcasting mechanism. In particular, method selection is based solely on the dynamic type of the arguments; there is no way to pass the desired types of upcast arguments.

To support upcasting, the MultiJava compiler creates an additional statically overloaded method for each regular generic function entry point (i.e., for each method created by the translation of Section 5.1, and for each apply method created by the translation of Section 5.2). This alternative entry point takes all the original arguments of the generic function plus additional class object arguments for the receiver and each original argument of class type. These additional arguments signal the desired classes for the original arguments and are used to select the appropriate multimethod code to execute.

Figure 13 shows the translation, enhanced to support upcasting, of the non-encapsulatable multimethods of Figure 9. The following code using an upcast:

```java
Shape s = new Triangle();
s = ((Shape)s).rotate(90.0);
```

is compiled to:

```java
Shape s = new Triangle();
s = Rotate_rotate_anchor .function.apply(s,90.0,Shape.class);
```

The simple translation given in Figure 13 duplicates the code for each multimethod body, including it in the regular generic function code and again in the method for handling upcast arguments. A more efficient scheme would factor out each method body into a separate static method, which is merely invoked from each of the two dispatcher functions.

An upcast can be from one multimethod in a dispatching function to another, less specific multimethod within the same dispatching function. Such cases can be identified statically when compiling the dispatching function and implemented more efficiently. A call from one part of a compiled method to another can be implemented without code duplication using the 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 normal dispatching and upcasts).

A final optimization applies to upcasts to encapsulatable generic functions where only the receiver is being upcast. In this case, an invokespecial bytecode can be used to call directly (without dispatching by the Java virtual machine) the regular dispatching function of the class to which the receiver has been upcast, avoiding the alternative upcast dispatching function. Regular Java super sends are a special case of this optimization, and hence are compiled just as in regular Java. The tricky case of a regular Java super send from within the inner class generated by a non-encapsulatable method is handled by exploiting the fact that JVMs do not require the target of invokespecial to be a superclass of the sender.

### 6. DISCUSSION

Our MultiJava proposal would necessitate straightforward extensions to the reflection API in Java. We leave these changes as future work.

In Java, one can declare that a method is final, which prevents it from being overridden. Similarly, the MultiJava type system can prevent a multimethod $M$ that is declared to be final from being overridden, in the sense that there can be no other multimethod in
the same generic function whose tuple of specializers is a pointwise subtype of \( M \)'s. Restriction \( R1 \) allows this condition to be easily checked. That is, it ensures that the multimethods that a particular multimethod \( M \) overrides are all available when \( M \)'s compilation unit is compiled.

One addition to MultiJava that we considered is the ability to extend interfaces, using an \texttt{extend interface} declaration. However, such a declaration would not be safe with MultiJava's type system. The problem is that, due to restriction \( R3 \), one would need to implement the added method to avoid the problem demonstrated in Figure 6. However, there is no way to implement a method in an interface.

Our original plan for MultiJava was to apply the concept of multiple dispatch as dispatch on tuples [Leavens & Millstein 98]. With this approach, all multimethods would be external to classes. Multimethods would be declared like

```java
public Shape apply(Shape this_, float a) {
    // the regular generic function entry point
    if (this_ instanceof Triangle) {
        Triangle this2_ = (Triangle) this_;
        /* method 3 body, substituting this2_ for this */
    }
    else if (this_ instanceof Rectangle) {
        Rectangle this2_ = (Rectangle) this_;
        /* method 4 body, substituting this2_ for this */
    }
    else {
        /* method 5 body, substituting this_ for this */
    }
}

public Shape apply(Shape this_, float a, Class this_Class) { // the upcasting entry point
    if (Triangle.class.isAssignableFrom(this_Class)) {
        Triangle this2_ = (Triangle) this_;
        /* method 5 body, substituting this2_ for this */
    }
    else if (Rectangle.class.isAssignableFrom(this_Class)) {
        Rectangle this2_ = (Rectangle) this_;
        /* method 4 body, substituting this2_ for this */
    }
    else {
        /* method 3 body, substituting this_ for this */
    }
}
```

Figure 13: Translation of non-encapsulatable multimethods with upcasting support.

However, the separation of arguments into dispatched and non-dispatched sets also brings several problems. The tuple approach does not provide for robust client code, because if a method is modified so that different arguments are dispatched upon, then all clients of that method must be changed, since the syntax of calls changes. With MultiJava client source code and compiled code is insensitive to the set of arguments a generic function dispatches upon. The tuple approach 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 simply dispatched methods, which includes all existing Java code. MultiJava does not have this restriction. Indeed, MultiJava is strictly more expressive than the tuple approach—the tuple approach could be added as syntactic sugar in MultiJava.

7. RELATED WORK

Most previous work on statically-typed multimethods whose arguments are treated symmetrically has been based on languages originally designed to support multimethods or on simple core languages. The latter is the case for the Tuple idea [Leavens & Millstein 98] discussed above. Our work addresses issues arising in the context of extending an existing, full-featured language.

Encapsulated multimethods [Castagna 95, Bruce et al. 95] are a design for adding 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 encapsulatable generic functions
yields compiled code similar to what would arise from encapsulated multimethods, but we hide the asymmetry of dispatch from programmers.

Boylan and Castagna demonstrated the addition of asymmetric multimethods to Java using “parasitic methods” [Boylan & Castagna 97]. To avoid the then-unsolved modularity problems with symmetric multimethods, their implementation is based on the idea of encapsulated multimethods. Parasitic multimethods overcome the limitations of encapsulated multimethods by supporting a notion of multimethod inheritance and overriding. Parasitic multimethods 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 further complicated by rules based on the textual order of multimethod declarations. Additionally, overriding parasitic methods must be declared as parasites, which in effect adds $\oplus$ 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 extensible 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 extensible 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 extended classes is required if aspects are changed. MultiJava’s extensible class technique is a key element of aspect-oriented programming; MultiJava brings this technique to the Java community while maintaining modular static typechecking and compilation.

8. FUTURE WORK AND CONCLUSIONS

In this paper we have shown how to extend Java with multimethods and extensible classes. 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), to support upcasts, and to be compiled efficiently, modularly, and interoperably with existing Java code.

There are several possible areas for future work. We plan to complete an implementation of our design before the final OOPSLA deadline. In addition, one might consider compilation strategies that allow one to declare new fields and constructors in \texttt{extend class} declarations. Another area for future work is to compare the usability of MultiJava with the tuple approach [Leavens & Millstein 98].

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-9455776, 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.

10. REFERENCES


