MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch

Curtis Charles Clifton

TR #01-10
November 2001

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.

2001 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented programming; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages, multiparadigm languages, MultiJava; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, classes and objects, control structures, inheritance, modules, packages, patterns, procedures, functions, and subroutines, multimethods, generic functions, open classes, pleomorphic methods; D.3.4 [*Programming Languages*] Processors — compilers.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This paper describes the design, implementation, and evaluation of MultiJava, a backward-compatible extension to The Java Programming Language™ 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. Multiple dispatch allows the method invoked by a message send to depend on the run-time types of any subset of the argument objects. MultiJava is the first full-scale programming language to support these features while retaining modular static typechecking and compilation.

The paper defines the notions of modular editing, typechecking, and compilation, and describes two problems, the augmenting method problem and the binary method problem, that heretofore had not been solved in a modular way. We describe the architecture and key implementation details of our MultiJava compiler, `mjc`. `mjc` is open-source and is freely available for downloading. We present an evaluation of MultiJava that demonstrates the ease of extending code written in the language. We also provide empirical results for the performance of MultiJava versus the previous partial solutions to the augmenting method and binary method problems. These results demonstrate that MultiJava's performance is comparable to that of the partial solutions, while the language provides full solutions to the problems.

SECTION 1. INTRODUCTION

This paper describes the design, implementation, and evaluation of MultiJava, a backward-compatible extension to The Java Programming Language™ [Gosling et al. 2000, Arnold et al. 2000] 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 1998, Millstein and Chambers 1999]. *Multiple dispatch* allows the method invoked by a message send to depend on the run-time types of any subset of the argument objects; this is in contrast to *single dispatch* where the method invoked depends on the run-time type of a distinguished receiver object and the *compile-time* types of the other argument objects. Subsection 1.1 below provides a more in-depth discussion of open classes, multiple dispatch, and what it means for multiple dispatch to be symmetric.

Many of the key ideas in this paper were first introduced by the author, with Leavens, Chambers, and Millstein in a paper for OOPSLA 2000 [Clifton et al. 2000], hereafter referred to as CLCM2000. The question addressed there, and here as well, is how to scale the theoretical results of the Dubious language [Millstein and Chambers 1999] to a full-scale language, thus providing open classes and symmetric multiple dispatch while maintaining modular, static typechecking and compilation. The contributions of our OOPSLA 2000 paper include:

- the first demonstration of open classes and symmetric multiple dispatch in a full-scale programming language with modular, static typechecking and compilation,
- a novel compilation scheme that is modular (each class or set of added methods can be compiled separately) and efficient (additional run-time cost is incurred only when the new features are actually used), and
- a language design that extends Java while retaining backward-compatibility and interoperability with existing Java source code and bytecode.

The remainder of this introduction is divided into two subsections. The first introduces two of the problems solved by open classes and multiple dispatch and a third problem that has kept these features from being successfully combined with modular, static typechecking and compilation. It also discusses what we mean by the term “modular”. Following the background information, another subsection highlights the key contributions of this work.

1.1. Background

MultiJava adds open classes and symmetric multiple dispatch to Java while maintaining modular static typechecking and compilation. This subsection provides background and motivation for these features by describing two of the problems that are solved by open classes and multiple dispatch.

Modularity is a theme that runs throughout this discussion, so it is helpful to begin by describing it in some detail. We say that an operation (like editing, typechecking, or compilation) performed on a program is *modular* if the operation can be achieved by manipulating a proper subset of the program's definition. Thought of in this way, modularity is more a continuum than an absolute; that is, the smaller the subset that is manipulated, the more modular the operation. To simplify discussion we will use the following notions:

- We say that editing of a program is modular if it does not require changing preexisting code. In Java, editing which introduced new *compilation units* would be consider modular[1]; editing which required changing or adding to preexisting compilation units would not be modular.
- We say that typechecking (or compilation) is modular if a single compilation unit can be typechecked (or compiled) without typechecking (or compiling) the code of other compilation units. Only the type signatures of the declarations in other compilation units are required for modular typechecking (or compilation) [Cardelli 1997].
- We say that a language is a *modular, statically-typed language* if it supports modular static typechecking and modular compilation. In such a language modular editing would require the typechecking and compilation of only the additional compilation units added to the program.

This subsection begins by describing two problems, the augmenting method problem and the binary method problem, that are solved by open classes and multiple dispatch. We then briefly describe the modularity problem that has, until now, prevented the successful integration of these features into a modular, statically-typed programming language.

1.1.1. Augmenting Method Problem

A well-known challenge for object-oriented languages is reconciling the addition of new sub-classes to a class hierarchy with the addition of new operations to existing classes [Reynolds 1975, Cook 1991, Odersky and Wadler 1997, Krishnamurthi et al. 1998, Findler and Flatt 1999, Zenger and

1. A Java compilation unit corresponds to a single file in Java implementations based on file systems, see §7.6 of [Gosling et al. 2000].

Odersky 2001]. For maintenance reasons the addition of new classes and new operations should be done in a modular way—without requiring any modifications to existing code. This is not possible in existing single dispatch object-oriented languages, like C++ [Stroustrup 1997], Smalltalk [Goldberg and Robson 1983], Java and others.

One can view the classes of a class hierarchy, or more generally the types in a collection of related types, as columns in a matrix. The rows in the matrix represent the operations available on the types. Table 1 shows such a matrix for a type `Shape` and a subtype `Rectangle`.

Table 1: A matrix view of types representing shapes

	Types	
Operations	`Shape`	`Rectangle`
`topBound()`	return the y-coordinate of the top-most point	return the y-coordinate of the top edge
`leftBound()`	return the x-coordinate of the left-most point	return the x-coordinate of the left edge
`shrink()`	mutate the `Shape` by moving all points half the distance towards the top, left bound	mutate the `Rectangle` by setting width and height to half the original values

The cells in the table describe the results of invoking the given operation on an instance of the given type. Procedural and object-oriented programming provide complementary support for extending a matrix such as this. In procedural programming the code is generally organized based on the rows of the table. For example a single `topBound` function might be implemented using a variant-case expression[2] in Scheme [Friedman et al. 1992]:

```
(define topBound
    (lambda (s)
        (variant-case s
            (Shape (xValues yValues) (max yValues))
            (Rectangle (top left height width) top)
            (else (error “unknown type”)))))
```

A separate function is implemented for each operation. Using this technique it is a simple matter to add a new operation, for example `circumference`. The new operation can be thought of as a new row in

2. Briefly, the `variant-case` in the sample code dispatches on the type of `s`. Only one of the cases within the variant-case will be executed. For example, if `s` is an instance of `Shape` then the result of the variant-case is the result of evaluating the body of the Shape case, i.e., `(max yValues)`, with the two variables, `xValues` and `yValues`, bound to the like-named fields of `s`.

the matrix. A single new function is written to encode the new operation, again using the variant-case technique:

```
(define circumference
    (lambda (s)
        (variant-case s
            (Shape (xValues yValues) ...)
            (Rectangle (top left height width) (* 2 (+ height width)))
            (else (error "unknown type")))))
```

Unfortunately, this approach does not support the modular addition of new types. For example, if one wishes to add a type for circles to the matrix, one must edit the definitions of every existing operation, such as `topBound` and `circumference`, to add a case for the new type.

By contrast, with object-oriented programming the code is organized based on the types. A single class is written for each column in the matrix. For example, in Java the `Rectangle` type might be implemented as follows:

```
public class Rectangle extends Shape {
    /* private fields */
    private double top, left, height, width;

    /* constructor */
    public Rectangle( double top, double left, double height, double width) {
        super( /* pass four corner points of the rectangle to Shape's constructor */ );
        this.top = top;         this.left = left;
        this.height = height;   this.width = width;
    }

    /* methods described in Table 1 */
    public double topBound() { return top; }
    public double leftBound() { return left; }
    public void shrink() {
        height = height / 2;    width = width / 2;
    }
}
```

With the object-oriented approach it is simple to handle the addition of new types to the matrix. One simply writes a new class implementing all the operations from the matrix for the new type. Thus the object-oriented approach succeeds where the procedural approach fails. Unfortunately, the converse is also true; the object-oriented approach does not support the addition of new operations in a modular way. For example, to add the `circumference` operation one would have to edit each existing class to add the new operation. We call this need for non-modular editing to add a new operations or new classes the *augmenting method problem.*

One solution to the augmenting method problem is to add the new operation by writing new sub-classes for each of the existing classes. The subclasses would include code for the new operation and would inherit the code for the remaining operations. Unfortunately, existing client code would still

include code to instantiate the original classes, so the client code would have to be modified to create instances of the new subclasses instead [Hölzle 1993]. If these new instances were assigned to variables of the existing types, then an explicit downcast would be required to access the new methods. Also if instances of the original classes were stored in a database, a non-modular conversion of the database would be required to convert the original classes to new subclasses.

A second approach is to use the visitor design pattern [Gamma et al. 1995] (pp. 331–344), which evolved 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 version of the `Shape` class hierarchy in Figure 1a; here the classes are augmented with an `accept` method according to the visitor pattern. Operations on shapes are structured in their own class hierarchy, each operation becoming a subclass of an abstract `ShapeVisitor` class as shown in Figure 1b. The client of an operation on shapes invokes the `accept` method of a shape, passing a `ShapeVisitor` instance representing the operation to perform; in this case, mutating the shape by flipping it about its vertical axis:

```
someShape.accept(new YAxisFlipVisitor(…))
```

The `accept` method of each kind of shape then uses double-dispatching [Ingalls 1986] to invoke the method of the visitor that is appropriate for that shape.

The main advantage of the visitor pattern is that new operations can be added modularly, without needing to edit any of the `Shape` subclasses: the programmer simply defines a new `ShapeVisitor` subclass containing methods for visiting each class in the `Shape` hierarchy. However, use of the visitor pattern brings several drawbacks, including the following, listed in order of 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 `Shape` class is first implemented. For example, had the `Shape` hierarchy not been written with an `accept` method, which allows visits from the `ShapeVisitor` hierarchy, it would not have been possible to add the horizontal-flipping 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; for example Figure 1c shows how results must be stored in the state of the visitor. To use this result-returning visitor requires code like:

a)
```
public class Shape {
    /* constructor and field declarations omitted */
    /* ... */
    public void accept( ShapeVisitor v ) {
        v.visitShape( this );
    }
}
public class Rectangle extends Shape {
    /* constructor and field declarations omitted */
    /* ... */
    public void accept( ShapeVisitor v ) {
        v.visitRectangle( this );
    }
}
```

---

b)
```
public abstract class ShapeVisitor {
    /* ... */
    public abstract void visitShape( Shape s );
    public abstract void visitRectangle( Rectangle r );
    /* abstract methods for other Shape subclasses */
}
public class YAxisFlipVisitor extends ShapeVisitor {
    /* ... */
    public void visitShape( Shape s ) {
        yFlipAbout( centroid(s.borderPoints()), s.borderPoints() );
    }
    public void visitRectangle( Rectangle r ) { /* does nothing */ }
}
```

---

c)
```
public class CircumferenceVisitor extends ShapeVisitor {
    private double result;
    public double getResult() { return result; }
    /* ... */
    public void visitShape( Shape s ) {
        result = accumDistance(s.borderPoints());
    }
    public void visitRectangle( Rectangle r ) {
        result = 2 * (r.height() + r.width());
    }
}
```

Figure 1: Java code for some participants in the visitor design pattern:
part a) shows some classes of the Shape hierarchy augmented with accept methods,
part b) shows some classes of the `ShapeVisitor` hierarchy that implement operations on shapes,
part c) shows an operation that returns a result through the visitor's state

```
CircumferenceVisitor v = new CircumferenceVisitor();
someShape.accept( v );
System.out.println( "The circumference is " + v.getResult() );
```

- 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 `Shape` classes in a modular way. For example, if a new `Shape` subclass were introduced, the `ShapeVisitor` class and all subclasses would have to be modified to contain a method for visiting the new kind of node. Thus, visitor trades the non-modularity of the object-oriented approached for the non-modularity of the procedural approach [Krishnamurthi et al. 1998]. Proposals have been advanced for dealing with this well-known limitation [Krishnamurthi et al. 1998, Martin 1998, Nordberg 1998, Vlissides 1999, Zenger and Odersky 2001], but they suffer from additional complexity (in the form of hand-coded typecases, more complex class hierarchies, and factory methods) that make them even more difficult and error-prone to use. More details on this related work are discussed below.

The paper by Krishnamurthi, et al. briefly describes a tool, Zodiac, to automate the generation of their more complex “extensible visitor” code [Krishnamurthi et al. 1998], however it seems that the tool must be used from initial code development; it is not possible to use Zodiac to extend existing code in a modular way.

Zenger and Odersky describe a Java language extension that introduces extensible algebraic datatypes and a typecase construct. They describe a technique called “extensible algebraic datatypes with defaults” that is more powerful than the extensible visitor pattern and seems to solve the augmenting method problem. This language extension is discussed further in Section 7, but we note here that, like Zodiac, the extended language features must be used from initial development.

Open classes, as described in Subsection 2.1, provide a more general solution to the augmenting method problem by allowing new methods to be added to a class without editing the class directly and without prior planning on the part of the original class designer.

1.1.2. Binary Method Problem

The well-known binary method problem is caused by traditional object-oriented languages’ reliance on single dispatch [Bruce et al. 1995]. Before describing the problem in detail it is helpful to introduce some terminology.

Though there are some discrepancies in the literature, we will use the term *dispatch* to refer to the selection, *at run-time*, of the appropriate method to invoke in response to a message send. Dispatch in

object-oriented languages can be divided into single and multiple dispatch. Multiple dispatch is found in Common Lisp [Steele Jr. 1990, Paepcke 1993], Dylan [Shalit 1997, Feinberg et al. 1997], and Cecil [Chambers 1992, Chambers 1995]. It 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, as noted above, 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* (i.e., compile-time) types of the arguments influence method selection via static overload resolution; the *dynamic* (i.e., run-time) types of the non-receiver arguments are not involved in method dispatch.

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. Symmetric multiple dispatch is used in Cecil, Dylan, Kea [Mugridge et al. 1991], the λ&-calculus [Castagna et al. 1995, Castagna 1997], $ML_{\leq}$ [Bourdoncle and Merz 1997], and Tuple [Leavens and Millstein 1998].

The restriction to single dispatch in languages like Java and C++ is sometimes 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. 1995]. In the portion of the `Shape` class given in Figure 2a, the method for checking whether two shapes overlap is a binary method. But when comparing two rectangles, one can use a more efficient overlap-detection algorithm than when comparing arbitrary shapes. The first way one might attempt to implement this idea in a Java program is given in Figure 2b.

Unfortunately, this binary method for two `Rectangle`s does not provide the desired semantics. In particular, the new `overlaps` method cannot be safely considered to override the original overlap-detection method, because it violates the standard contravariant typechecking rule for functions [Cardelli 1988]: the argument type cannot safely be changed to a subtype in the overriding method. To wit, suppose the new method were considered to override the `overlaps` method from class `Shape`. Then, by single dispatch, a method invocation `s1.overlaps(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` overlap-detection method when `s2` is an arbitrary `Shape`, even though the method expects its argument to be another `Rectangle`. This would cause a run-time type error, when `Rectangle`'s method calls the `containsCornerOf` utility method that tries to access the fields in its argument `r` that are not inherited from `Shape`.

a)
```
public class Shape {
    /* ... */
    public boolean overlaps(Shape s) {
        Iterator pairs = orderedPairsOf(s.borderPoints());
        while (pairs.hasNext()) {
            if ( ((Pair)pairs.next()).lineSegment().crosses( this ) ) {
                return true;
            }
        }
        return false;
    }
}
```

b)
```
public class Rectangle extends Shape {
    /* ... */
    public boolean overlaps(Rectangle r) {
        return containsCornerOf(r);
    }
    private boolean containsCornerOf(Rectangle r) {
        return contains(r.topLeft) || contains(r.topRight) ||
               contains(r.bottomLeft) || contain(r.bottomRight);
    }
}
```

Figure 2: Binary method examples:
part a) gives a binary method on two `Shape` objects,
part b) gives a binary method on two `Rectangle` objects.

To resolve the type safety problem of Figure 2b, Java, like C++, considers `Rectangle`'s `overlaps` method to *statically overload* `Shape`'s method.[3] Conceptually, one can think of each method in a program 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.[4] Statically overloaded methods belong to distinct generic functions, just as if the methods had different names. At compile-time 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 `overlaps` methods belong to different generic functions, and Java determines statically which generic function is invoked for any `overlaps` message send site based on the *static* type of the message argument expressions. For example, consider the client code in Figure 3. Although the objects passed as arguments in the four `overlaps` message sends are identical,

3. The details of static overloading in C++ are more complex than those for Java.
4. These terms are defined more formally in Subsection 2.1.2 on page 18.

```
Rectangle r1, r2;
Shape s1, s2;
boolean b1, b2, b3, b4;

r1 = new Rectangle( /* ... */ );
r2 = new Rectangle( /* ... */ );
s1 = r1;
s2 = r2;

b1 = r1.overlaps(r2);
b2 = r1.overlaps(s2);
b3 = s1.overlaps(r2);
b4 = s1.overlaps(s2);
```

Figure 3: Code demonstrating the “unnatural” semantics of static overloading

these message sends do not all invoke the same method. In fact, only the first message send will invoke the `Rectangle` overlap-detection method. The other three messages will invoke the `Shape` overlap-detection method, because the static types of these arguments cause Java to bind the messages to the generic function introduced by `Shape`’s `overlaps` method. Likewise, the first message is statically bound to the generic function introduced by `Rectangle`’s `overlaps` method. Thus Java’s static overloading solves the type-safety problem, but provides an “unnatural” semantics.[5]

This demonstrates the *binary method problem* of single dispatch languages—because they cannot safely use subtypes in the argument positions of overriding methods, single dispatch languages cannot easily specify overriding binary methods for cases when both the receiver and non-receiver arguments are subtypes of the original types; furthermore, single dispatch languages cannot easily take advantage of the private representation of the non-receiver argument.

In Java, one can solve the binary method problem by performing explicit run-time type tests and associated casts; we call this coding pattern a *typecase*.[6] This is the basic technique of encapsulated multimethods [Bruce et al. 1995, Castagna 1995]. For example, one could implement the `Rectangle` overlap-detection method as follows:

```
public class Rectangle extends Shape {
    /* ... */
    public boolean overlaps( Shape s ) {
        if( s instanceof Rectangle ) {
            Rectangle r = (Rectangle) s;
            return containsCornerOf(r);
        } else {
            return super.overlaps(s);
        }
```

5. This claim of unnaturalness is supported by the fact that many of the computer scientists to whom we have described this example required a demonstration to be convinced of the semantics.
6. The typecase pattern is analogous to the variant-case code in Scheme introduced on page 3.

```
    }
    /* ... */
}
```

This version of the `Rectangle` overlap-detection method has the desired semantics. In addition, since it takes an argument of type `Shape`, this method can safely override `Shape`'s `overlaps` method, and is part of the same generic function. All message sends in the example client code above will now invoke `Rectangle`'s `overlaps` method.

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

A related partial 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 on page 6). With double-dispatching, instead of using an explicit `instanceof` test to find out the run-time type of the non-receiver argument `s`, as in the typecase example, this information is obtained by performing a second message send, as in Figure 4. This second message is sent to the non-receiver argument of the original message, but with the name of the message encoding the dynamic class of the original receiver. This can be seen in the `overlaps` methods of the figure.

Double-dispatching avoids the need for an explicit typecase over all the possible argument shapes in every subclass. Instead it reuses the language's built-in method dispatching mechanism. However, double-dispatching is even more tedious to implement by hand than typecases, requiring $n(n+1)$ methods to handle all pairs of n types. Also, 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. (This modification introduces the generic function for the new subclass, akin to `overlapsRectangle` in our example.)

Multimethods provide an elegant solution to the binary method problem by supporting safe covariant overriding in the face of subtype polymorphism [Castagna 1995, Bruce et al. 1995]. This is demonstrated in Subsection 2.2 below. Multimethods also provide a more uniform and expressive approach to overload resolution by eliminating the need for static overloading.

```
public class Shape {
    /* ... */
    public boolean overlaps( Shape s ) {
        s.overlapsShape( this );
    }
    protected boolean overlapsShape( Shape s) {
        Iterator pairs = orderedPairsOf(s.borderPoints());
        while (pairs.hasNext()) {
            if ( ((Pair)pairs.next()).lineSegment().crosses( this ) ) {
                return true;
            }
        }
        return false;
    }
    protected boolean overlapsRectangle( Rectangle r ) {
        // no special code so...
        return overlapsShape( r );
    }
}

public class Rectangle extends Shape {
    /* ... */
    public boolean overlaps( Shape s ) {
        s.overlapsRectangle( this );
    }
    protected boolean overlapsRectangle( Rectangle r ) {
        return containsCornerOf(r);
    }
    /* ... */
}
```

Figure 4: Examples of binary methods encoded using double-dispatch

1.1.3. The Modularity Problem

If a language can solve the augmenting and binary methods problems by supporting open classes and multimethods, then why aren't they widely supported? One reason is that both features have suffered from a *modularity problem* [Cook 1991]: independently-developed modules, which typecheck in isolation, may cause type errors when combined.[7] Object-oriented languages without multimethods and open classes do not suffer from the modularity problem; for example, in Java, one can safely typecheck each compilation unit in isolation. Because of the modularity problem, previous work on adding multimethods to an existing statically-typed object-oriented language has either forced global typechecking [Leavens and Millstein 1998, Dutchyn et al. 2001] or has employed asymmetric multiple

7. We elaborate on this modularity problem in Subsection 3.4, after we have introduced sufficient syntax to describe it in greater detail.

dispatch in order to ensure modularity [Boyland and Castagna 1997]. We will see that MultiJava solves the modularity problem by a few simple restrictions to the expressiveness of the language.

1.2. Goals and Contributions

In [Millstein and Chambers 1999] the authors present a simple core language, Dubious, with open classes and symmetric multiple dispatch. The authors describe several different sets of restrictions that permit modular static typechecking of Dubious while still providing the flexibility necessary to solve the augmenting and binary methods problems. In MultiJava we apply the most modular of the Dubious type systems, System M, to the Java language.[8] The result is the first full-scale programming language to support open classes and symmetric multiple dispatch with modular, static typechecking and compilation.

The design of MultiJava is predicated on the following goals and constraints:

- MultiJava must provide complete backward compatibility with the extant Java language. Code written in Java must have the same semantics when compiled with a Java compiler or a MultiJava compiler, including code that relies on Java's static overloading. It must be possible to extend existing Java classes via the open class mechanism. It must be possible to override existing single dispatch methods with multimethods.
- The modular static typechecking and compilation properties of Java must be maintained.
- To allow for wide use of code written in MultiJava, output of the MultiJava compiler will target the standard *JVM*, or Java Virtual Machine.
- For regular Java code the bytecode produced by the MultiJava compiler should be no less efficient than that generated by a standard Java compiler. Since MultiJava source code using open classes or multiple dispatch cannot easily be expressed in regular Java, the efficiency of the generated code for these features is not a primary concern. However, the bytecode should have efficiency comparable to standard Java code using the extensible visitor pattern, hand-coded double-dispatching, typecases, or other partial solutions.

By satisfying the first two constraints MultiJava solves the augmenting method and binary method problems in situations where development extends existing libraries. The first constraint says

8. System M achieves its modularity by sacrificing some expressiveness. We highlight some of the idioms that are disallowed when we introduce the restrictions. In Subsection 6.1.7 we discuss how we might adopt the more expressive System E for use in MultiJava.

that our solution to the augmenting method problem provides modular editing; the second says that our solution provides modular typechecking and compilation. Of course solving these problems in situations where development extends existing libraries means that these problems are solved in situations where development proceeds from scratch.

By satisfying the last two constraints MultiJava is more than an academic curiosity. The language suits any circumstance where Java would be suitable, and perhaps other circumstances due to its greater extensibility and expressiveness. One might argue that since the number of Java programmers is large and the number of MultiJava programmers is small (thus far), that a project is better served using Java. We think the syntax and semantics, as presented in Section 2, are so “Java-like” that the experienced Java developer will have no problems adopting MultiJava.

To assist the reader, much of the content from CLCM2000 is repeated in this paper. The discussion is explicit about the material that is new with this work. Here is a brief outline of the material that follows, with the key new contributions in each section noted:

Section 2 describes the syntax and semantics of the MultiJava language both by example and formally where necessary for understanding. This section also describes the typechecking strategy for MultiJava and the restrictions necessary for modularity. While much of this section comes from CLCM2000, it also provides several new contributions including:

- a discussion of methods that belong to more than one generic function and the introduction of a term to describe such methods,
- support for run-time dispatching on array types and overloading of external generic functions,
- additional restrictions on the use of some language features that, while not necessary for soundness, provide software engineering benefits, and
- a thorough treatment of upcalls[9] that identifies and remedies an encapsulation problem not previously discussed and that classifies upcalls into two kinds, superclass method invocations and overridden method invocations, with distinct syntax and semantics.

Section 3 describes the translation of MultiJava source code into regular Java bytecode. This section contributes compilation strategies for the two kinds of upcalls, for private methods written using the open class syntax, and for encoding MultiJava specific information in bytecode for efficient retrieval by MultiJava compilers.

9. We say a method invocation is an *upcall* if it targets a method in a superclass of the calling method or it targets a method overridden by the calling method. The invocation `super.toString()` in Java is an example.

The remaining sections of the paper are all new with this work, except for a brief discussion of the alternative language design, TupleJava, which comes from CLCM2000. Section 4 describes the architecture of our MultiJava compiler and highlights some of the interesting techniques used in implementing it. Section 5 presents an evaluation of the language by describing an application implemented using MultiJava and by comparing performance results on code examples implemented first in MultiJava and then using several of the partial solutions identified above. In Section 6 we describe some possible extensions to MultiJava and review TupleJava. Section 7 discusses related work and Section 8 concludes.

SECTION 2. LANGUAGE DESIGN

We solve the augmenting and binary methods problems by including open classes and multiple dispatch in MultiJava. This section describes how we add these features to Java while solving the modularity problem, thus retaining Java's static modular typechecking and compilation. Subsection 2.1 describes MultiJava's open class feature, while Subsection 2.2 describes the syntax and semantics of multiple dispatch in MultiJava. In Subsection 2.3 we discuss the interaction of these features. With the necessary syntax introduced, Subsection 2.4 elaborates on the modularity problem that was introduced above and describes the restrictions imposed by the type system to solve this problem.

2.1. Open Classes in MultiJava

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. Contrary to the visitor pattern, it does this in a way that allows new subclasses to be introduced modularly. Thus MultiJava's open classes solve the augmenting method problem.

2.1.1. Declaring and Invoking Augmenting Methods

The key new language feature involved in open classes is the *augmenting method* declaration, whose syntax is specified in Figure 5 on page 17. Using augmenting methods, the functionality of the circumference-calculating visitor from Figure 1c can be written as in Figure 6.

A program may contain several augmenting method declarations that add methods to the same class; for example the use of the `circumference` generic function in a program would not preclude the use of a separate `area` generic function declared using augmenting method declarations. As in Java, the bodies of augmenting methods may use the keyword "`this`" to reference the receiver object. Also as in Java the use of "`this`" is only required when passing a receiver reference to another method or when accessing a field of the receiver that is hidden by a local parameter, type, or variable declaration. Otherwise field references and method calls implicitly target the receiver object. See, for example, the calls to `width()` and `height()` in the `circumference` augmenting method for `Rectangle` in Figure 6. It is permissible to use `this` in situations where it is not required. For example, we could have written `this.width()` in this example.

Clients invoke augmenting methods exactly as they would the class's original methods. For example, the `circumference` method of `someShape` is invoked as follows:

```
someShape.circumference()
```

CompilationUnit[7.3]:
    PackageDeclaration[7.4.1]$_{opt}$ *ImportDeclarations*[7.5]$_{opt}$ *TopLevelDeclarations*$_{opt}$
TopLevelDeclarations:
    TopLevelDeclaration
    TopLevelDeclarations *TopLevelDeclaration*
TopLevelDeclaration:
    TypeDeclaration[7.6]
    AugmentingMethodDeclaration
AugmentingMethodDeclaration:
    AugmentingMethodHeader *MethodBody*[8.4.5]
AugmentingMethodHeader:
    MethodModifiers[8.4.3]$_{opt}$ *ResultType*[8.4] *AugmentingMethodDeclarator* *Throws*[8.4.4]$_{opt}$
AugmentingMethodDeclarator:
    ClassType[4.3] `.` *Identifier*[3.8] `(` *FormalParameterList*[8.4.1]$_{opt}$ `)`

Figure 5: Syntax extensions for MultiJava open classes:
This grammar extends the Java syntax given in the first 17 chapters of The Java Language Specification (distinct from the parser grammar given in chapter 18) [Gosling et al. 2000]. 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 bear superscript annotatations giving the pertinent section numbers from the Java specification.

```
// compilation unit "circumference"
package thesis.examples;
// Methods for calculating circumference
public double Shape.circumference() {
    return accumDistance(borderPoints());
}
public double Rectangle.circumference() {
    return 2 * (width() + height());
}
/* circumference calculation methods for other subclasses */
```

Figure 6: Circumference-calculating generic function using augmenting methods

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

2.1.2. Generic Functions, External and Internal

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

Recall that a generic function is a collection of methods consisting of a top method and all of the methods that (dynamically) override it. Conceptually, one can think of each method in a program (whether augmenting or declared within a class) as implicitly belonging to a generic function. For example, the `circumference` augmenting methods above introduce a single new generic function, providing implementations for two receiver classes. Each message send expression 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 an accessible method declaration M_{super} 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_{super}, hence M_{sub} overrides M_{super}. 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. It is also possible for a method to have two distinct top methods and thus belong to two generic functions even without open classes. This can happen when a single method of a subclass simultaneously overrides a superclass method and implements a method from an interface. We call such a method a *pleomorphic method*.[10] Pleomorphic methods present an interesting compilation challenge that was not considered in CLCM2000. This challenge and a solution are discussed in Subsection 3.3 on page 53.

A *reference type* in Java is any class type, interface type, or array type. We say that a reference type *S* is a *subtype* of a reference type *T* (equivalently, *T* is a *supertype* of *S*), and we write $S \leq: T$, if one of the following holds [Gosling et al. 2000] (§5.1.4):

- *T* is a class and *S* is either *T* or a class that extends *T*,
- *T* is an interface and *S* is a class that implements *T*,
- *T* is an interface and *S* is either *T* or an interface that extends *T*,
- *S* is an interface and *T* is `java.lang.Object`,
- *S* is an array type and *T* is `java.lang.Object`,
- *S* is an array type and *T* is `java.lang.Cloneable`,
- *S* is an array type and *T* is `java.io.Serializable`, or

10. Pleomorphic methods are named after the term in crystallography, since such methods can be considered points in two different type tuple lattices.

- *S* is an array type *S´*`[]`, *T* is an array type *T´*`[]`, and *S´* ≤: *T´*.

We say that *S* is a *proper subtype* of *T* if *S* ≤: *T* and *S* ≠ *T*. This subtype relation can be defined equivalently using the `instanceof` operator in Java: *S* is a subtype of *T* if, for all objects *s* of type *S*, the Java expression *s* `instanceof` *T* evaluates to `true`. The subtype relation used in this work is different than the relation given in CLCM2000 that did not consider array types.

We call a method declared via the augmenting method declaration syntax an *external method* if the class of its receiver is not declared in the same compilation unit. All other methods are *internal*. Besides methods declared in class declarations, this includes methods declared via the augmenting 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, an *external generic function* is one whose top method is external. All other generic functions are internal. Some methods of an external generic function can be internal methods (see Subsection 2.1.4 on page 20). The concepts embedded in our use of the terms internal and external are subtle. The reader should note the following:

- A regular Java method is always internal.
- An augmenting method may be internal (if it augments a class in the same compilation unit) or may be external.
- An internal method may belong to an internal or an external generic function.
- An external method always belongs to an external generic function (due to restriction ***R3*** presented in Subsection 2.4.2).

We will be rigorous in the use of the these terms in the remainder of this paper.

2.1.3. 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 thesis.examples.circumference;
```

will import the generic function `circumference` from the package `thesis.examples`. Similarly

```
import thesis.examples.*;
```

will implicitly import all the compilation units in the package `thesis.examples`, which will make all accessible (i.e., non-private) types and generic functions in that package available for use. Each compilation unit implicitly imports all the generic functions in its package.[11]

We call the set of methods and fields in a class the *signature* of that class. The set of methods and fields available to all clients of a class is the *public signature* of that class. The *apparent signature* of a class for a given client is the set of methods and fields available to that client. The explicit importation of external generic functions enables client code to manage the apparent signatures of the classes they manipulate. Only clients that import the `circumference` generic function will see the `circumference` operation in the apparent signature of `Shape`. Other clients will not have their apparent signatures for `Shape` polluted with this generic function. Furthermore, a compilation unit that did not import the existing `circumference` generic function could declare its own `circumference` generic function without conflict.

Java allows at most one public type (class or interface) declaration in a compilation unit.[12] This concession allows the implementation to find the file containing the source 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 a collection of overloaded public generic functions all with the same identifier. The CLCM2000 version of MultiJava allowed a file to contain the methods of just a single public generic function. But because the file name is derived from the generic function identifier, the original restriction prevented a package from overloading external generic functions on unrelated types or with different numbers of parameters. The new constraint permits such overloading while still allowing a compiler to easily locate imported generic functions.

2.1.4. Inheritance of Augmenting Methods

MultiJava extends Java's notions of method inheritance and subtype polymorphism to open classes. Because of this, a new subclass of `Shape` can be added without changing any existing code, as follows:

```
import thesis.examples.Shape;
import thesis.examples.circumference;
public class Parallelogram extends Shape {
    /* ... */
    public double circumference() {
        return 2.0 * (base() + side());
```

11. As of this writing, the implementation of this feature in `mjc` is not finished.
12. Java's restriction is somewhat more complex to account for its default access modifier, which gives access to all other classes in the package [Gosling et al. 2000] (§7.6).

```
// compilation unit "area"
import thesis.examples.Shape;
import thesis.examples.Rectangle;
public double Shape.area {
    return sumTriangles(sort(borderPoints()));
}
public double Rectangle.area {
    return width() * height();
}
```

Figure 7: Example used to show method inheritance for open classes

```
    }
}
```

A subclass can override any method in the apparent signature of its superclass, as in `Parallelogram`'s `circumference` method. Thus, unlike the visitor pattern, open classes permit the modular addition of new operations and new types. This method also illustrates a fact noted above—regular internal methods can be added to external generic functions.

Because of method inheritance, a client of `Parallelogram` can invoke any method in `Shape`'s apparent signature on an instance of `Parallelogram`, regardless of whether that method was visible in `Parallelogram`'s compilation unit. For example, suppose that a client program imported the `area` generic function from Figure 7. Even though the `area` method was not in the apparent signature of `Shape` from `Parallelogram`'s perspective, the client can still execute the following code:

```
Parallelogram par = new Parallelogram( /* ... */ );
double area = par.area();
```

This will execute the `Shape.area()` method from Figure 7, just as if `Parallelogram` had inherited the method from `Shape`.

Self-augmenting Classes

An interesting coding pattern with augmenting methods, and one not considered in CLCM2000, is the self-augmenting class. We say a class *C* is *self-augmenting* if *C* imports a generic function that includes a method with receiver type *C*. For example, if the `Rectangle` class were to import the `area` generic function of Figure 7, then `Rectangle` would be a self-augmenting. For self-augmenting classes *downcalls*, or calls from a superclass method to an overriding subclass method of a different generic function [Ruby and Leavens 2000], are potentially confusing. Consider the following code:

```
// compilation unit "size"
package thesis.operations;
public int Cell.size() {
```

```
    return 1;
}
// compilation unit "Cell"
package thesis.containers;
import thesis.operations.size;      // disallowed, see below
public class Cell {
    /* ... */
    public int capacity() {
        return size();
    }
}
```

In this code `Cell` is a self-augmenting class. The method call expression in the body of `Cell`'s `capacity` method calls the external generic function `size`. Now suppose that we declare a subclass of `Cell` as follows:

```
// compilation unit "Pair"
package thesis.containers;
public class Pair extends Cell{
    /* ... */
    public int size() {
        return 2;
    }
}
```

Since `Pair` does not import *size*, the method `size` is not in the apparent signature of `Cell` from `Pair`'s perspective. Thus the `size` method declared in `Pair` does not override the external method `Cell.size` but instead introduces a new generic function. The target generic function of an invocation is determined statically. Since the `size` method of `Pair` is not in the generic function targeted by `Cell`'s invocation of `size()`, `Pair.size` cannot be the target of the invocation. Therefore, an invocation of `capacity` on an instance of `Pair` will not result in a downcall to the `size` method of `Pair` as one might otherwise expect, but instead results in a call to `Cell.size`. Thus we have the following results:

```
new Cell().capacity()  →  1
new Pair().capacity()  →  1
```

To avoid this potential confusion we have chosen to disallow self-augmenting classes in the current version of the language. This doesn't seem to unduly restrict the expressiveness of the language, since if one were able to edit `Cell` to import *size*, then one could certainly write `size` inside of `Cell`. And if one wishes to separate concerns by keeping `size` in a distinct compilation unit, then one can always write the method `capacity` as an external generic function as well:

```
// compilation unit "capacity"
package thesis.operations;
import thesis.operations.size;
public int Cell.capacity() {
```

```
        return size();
    }
```

freeing `Cell` of the need to import *size*. Disallowing self-augmenting classes also avoids an encapsulation problem for superclass method invocations. A restriction for overcoming this encapsulation problem, discussed in Subsection 6.1 on page 91, would allow self-augmenting classes, though the restriction would do nothing to resolve the confusion that they can cause.

2.1.5. Encapsulation

MultiJava retains the same encapsulation properties as Java [Gosling et al. 2000] (§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.[13]

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.

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. An augmenting internal method has the same access privileges as a regular Java method, including the ability to access private members of its receiver class.

A consequence of these encapsulation properties of external methods is that one can view an external method as a composition of operations on the accessible fields and methods of the augmented class. This means, for example, that an external method cannot invalidate an object's invariant (assuming that doing so is impossible for regular Java clients).

2.1.6. Restrictions for Modular Typechecking

MultiJava's modular typechecking scheme is discussed in detail in Subsection 2.4 on page 34. We briefly survey the restrictions on external methods here.

13. 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 of the method's receiver class and also within the compilation unit in which they are introduced.

External methods may not be annotated as `abstract`, nor can they be added to interfaces. Suppose abstract external methods were allowed. A concrete subclass of the augmented abstract class could be declared in another compilation unit, without importing the abstract external method. Furthermore, the compilation unit declaring the abstract external method could not necessarily import all concrete subtypes of the augmented abstract class. Thus purely modular typechecking could not guarantee that the abstract external method was overridden for all concrete subclasses of the augmented abstract class. A similar argument holds for interfaces. However, this restriction on abstract external methods does not prohibit the declaration of concrete external methods that augment abstract classes.

A second consequence of modular typechecking is that an augmenting 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 augmenting methods in the same generic function with the same receiver class, leading to a clash. This requirement is thus necessary for solving the modularity problem.

2.1.7. Summary of Open Classes

A key benefit of open classes is that they solve the augmenting method problem. New client-specific operations can be written as external generic functions outside of the class declarations. Unlike with the visitor pattern, there is no need to plan ahead for adding the new operations. Each new operation can define its own argument types and result type, independently of other operations; there is no need to manipulate these through auxiliary fields. More importantly, open classes allow new subclasses to be added to the program modularly, because there is no visitor class hierarchy that needs to be updated.

Open classes also give programmers more flexibility in organizing their code. For example, the original circumference methods of Figure 6 can all be put in a single file separate from the compilation units defining the classes of the Shape hierarchy, supporting the separation of “cross-cutting” operations from the classes to which they belong, a key feature of subject-oriented and aspect-oriented programming [Harrison and Ossher 1993, Kiczales et al. 1997]. 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 library. New methods can even be added to a final class without violating the property that the class has no subclasses. Open classes also prevent the signature of a class from being “polluted” by methods that are specific to a particular program but not needed in others that use the class.

FormalParameter[8.4.1]:
    Type[4.1] `@` *ClassOrArrayType* *VariableDeclaratorId*[8.3]
    `...`
ClassOrArrayType:
    ClassType[4.3]
    ArrayType[4.3]

Figure 8: Syntax extensions for MultiJava multimethods:
See Figure 5 on page 17 for a description of the notation.

```
public class Rectangle extends Shape {
    /* ... */
    public boolean overlaps(Shape@Rectangle r) {
        return containsCornerOf(r);
    }
    /* ... */
}
```

Figure 9: Multimethod version of an overlap-detection method for two rectangles

2.2. Multiple Dispatch in MultiJava

In part to provide a clean and modular solution to the binary method problem, MultiJava allows programmers to write multimethods.

2.2.1. Declaring Multimethods

The syntax of our multimethod extension is specified in Figure 8.[14] Recall the `overlaps` binary method from Figure 2 on page 9. Using multimethods, an `overlaps` binary method for two `Rectangle` instances can be written as in Figure 9. This code is identical to the first solution attempt presented in Figure 2b, except that the type declaration of the formal parameter `r` is `Shape@Rectangle`[15] instead of simply `Rectangle`. The `Shape` part denotes the *static* type of the argument `r`. Consequently, the revised `overlaps` method belongs to the same generic function as `Shape`'s `overlaps` method from Figure 2a; 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 `overlaps` 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.

14. This syntax is modified from CLCM2000 to permit array types following the "@" terminal symbol.
15. Read "shape as rectangle".

2.2.2. Message Dispatch Semantics

To formalize the definition of MultiJava's message dispatch semantics we first introduce some terminology and then extend the definition of subtyping (introduced in Subsection 2.1.2 on page 18) to tuples of types.

In a formal parameter declaration, the type after an `@` symbol is referred to as the *explicit specializer* of the formal and we call the *S*`@`*T* construction a *specialized parameter type*. For a given method *M* with *n* non-receiver arguments, its *tuple of specializers* $(S_0, \ldots, S_n)$ is such that if *M*'s receiver type is *T*, then $S_0 =$ `@`*T* and, for $i \in \{1 \ldots n\}$, if *M* has an explicit specializer, U_i, at the *i*th position, then S_i is `@`U_i, otherwise S_i is the static type of the *i*th parameter. An `@`-sign preceding a type in the tuple of specializers indicates that the corresponding parameter is considered in dispatching decisions. The `Shape` class's `overlaps` method has the tuple of specializers (`@Shape`, `Shape`) while the `Rectangle` class's method has (`@Rectangle`, `@Rectangle`). The receiver type is always annotated with an `@`-sign since the receiver is always dispatched upon.

Subsection 2.1.2 introduced a subtyping relation for reference types. To describe MultiJava's dispatch semantics we must extend this definition to accommodate primitive types and the special `null` type. Furthermore our definition must recognize the inherent asymmetry in Java's dispatch semantics, where `null` may be passed in a non-receiver argument position, but a `null` value in the receiver position results in the familiar `NullPointerException`.

Recall that we write $S \leq: T$ if *S* is a subtype of *T* according to the subtype relation given in Subsection 2.1.2. Let $\leq_{\text{mic}}$ denote the subtyping relation induced by the method invocation conversion operation in Java [Gosling et al. 2000] (§5.3). Briefly, $S \leq_{\text{mic}} T$ if one of the following hold:

- *S* and *T* are reference types and $S \leq: T$,
- *S* = `null` and *T* is a reference type, or
- *S* and *T* are primitive types and *S* can be converted to *T* by widening primitive conversion [Gosling et al. 2000] (§5.1.2).

There are subtle differences between the $\leq:$ and $\leq_{\text{mic}}$ relations. The $\leq:$ relation applies only to reference types. But the $\leq:$ relation can be defined in terms of Java's `instanceof` operator, and a Java expression whose static type is a reference type may evaluate to `null`. The `instanceof` operator in Java is still applicable to such an expression and will evaluate to `false`. For any Java expression `e` with dynamic type *S*, if `e` evaluates to anything other than `null`, then `e instanceof` *T* implies $S \leq_{\text{mic}} T$. But if `e` evaluates to `null`, then `e instanceof` *T* is false but $S \leq_{\text{mic}} T$ is true.

We say that a type tuple $(S_0, \ldots, S_n)$ *is a subtype of* type tuple $(T_0, \ldots, T_n)$ if for each $i \in \{0 \ldots n\}$, one of the following holds

(***C1***) $S_i = U_i$, $T_i = V_i$, and $U_i \leq_{\text{mic}} V_i$

(***C2***) $S_i = U_i$, $T_i = \texttt{@}V_i$, and $U_i \leq: V_i$

(***C3***) $S_i = \texttt{@}U_i$, $T_i = V_i$, and $U_i \leq: V_i$

(***C4***) $S_i = \texttt{@}U_i$, $T_i = \texttt{@}V_i$, and $U_i \leq: V_i$

For multimethods the overrides relationship is extended to consider all argument positions. Let M_1 be a multimethod with tuple of specializers $(S_0, \ldots, S_n)$ and M_2 be a multimethod with tuple of specializers $(T_0, \ldots, T_n)$. M_1 *overrides* M_2 if M_1 and M_2 have the same name, number of arguments, and static argument types and $(S_0, \ldots, S_n)$ is a subtype of $(T_0, \ldots, T_n)$. In our overlaps example (`@Rectangle`, `@Rectangle`) is a subtype of (`@Shape`, `Shape`), because the predicate "`Rectangle` $\leq:$ `Shape`" satisfies ***C4*** for the receiver position and ***C3*** for the non-receiver position in the definition of subtyping for type tuples.

The semantics of message dispatch in MultiJava is as follows. For a message send $E_0.I(E_1,\ldots,E_n)$, 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 $(v_0, \ldots, v_n)$. To formalize the notion of *most-specific applicable method*, let $(C_0, \ldots, C_n)$ be the dynamic types of $(v_0, \ldots, v_n)$. A method with tuple of specializers $(S_0, \ldots, S_n)$ is *applicable* to $(v_0, \ldots, v_n)$ if $(C_0, \ldots, C_n)$ is a subtype of $(S_0, \ldots, S_n)$. The *most-specific* applicable method is the unique applicable method whose tuple of specializers $(S_0, \ldots, S_n)$ is a subtype of the tuple of specializers of every applicable method. If there are no applicable methods for a message send, a *message-not-understood* error occurs. If there are applicable methods but no unique most-specific one, a *message-ambiguous* error occurs. (Static typechecking, described in Section 2.4, can always detect and reject generic functions that could potentially cause such errors, solving the modularity problem.)

Given this dispatching semantics, the code in Figure 9 indeed solves the binary method problem. For example, consider an invocation `s1.overlaps(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 `Shape`'s and `Rectangle`'s `overlaps` 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.

```
public class Circle extends Shape {
    /* ... */
    public boolean overlaps(Shape s) {
        Iterator pairs = pairsOf(s.borderPoints());
        while (pairs.hasNext()) {
            if (line((Pair)pairs.next()).withinDistance(radius(),center()) {
                return true;
            }
        }
        return false;
    }
    public boolean overlaps(Shape@Rectangle r) {
        return  r.contains( center() ) ||
                r.withinDistance( radius(), center() );
    }
    public boolean overlaps(Shape@Circle c) {
        return distance( center(), c.center() ) < radius() + c.radius();
    }
}
```

Figure 10: MultiJava code demonstrating the mixing of methods with multimethods

The presence of an `@`-sign on a member of a tuple in a type tuple comparison indicates that the $\leq$: relation is used in that position; otherwise the $\leq_{mic}$ relation is used as in regular Java. For symmetry in multimethod dispatch, all arguments used in dispatching are compared using the same $\leq$: relation. Thus MultiJava's dispatching semantics is symmetric while naturally generalizing Java's dispatching semantics. If a MultiJava program uses no explicit 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 explicit specializers extends Java's normal dynamic dispatching semantics to these additional arguments.

2.2.3. 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 overlap-detection methods as in Figure 10 (signatures of the methods are in bold print).

All these methods have static argument type `Shape`, so they all are in the same generic function (introduced by the `overlaps` 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.overlaps(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 `overlaps` method. The third `Circle` method is most specific, so it is invoked. If `s1` is a `Circle` but `s2` is a `Shape`, 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 `Shape`'s `overlaps` method is applicable (since `Rectangle` only declares a method for pairs of `Rectangles`). Finally, if `s1` is a `Circle` and `s2` is `null`, then only the `Shape` method and the first `Circle` method are applicable. (Since it is not true that `null` ≤: `Circle` and it is not true that `null` ≤: `Rectangle`, by ***C2*** the last two `Circle` overlaps methods are not applicable. Since it is true that `null` $\leq_{mic}$ `Shape`, by ***C1*** the first `Circle` method is applicable, and similarly for the `Shape` overlaps method.)

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 2.4.) 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.

2.2.4. Other Uses of Multimethods

While binary methods are a commonly occurring situation where multimethods are valuable, other situations can benefit from multiple dispatch as well. For one example, consider a `display` generic function defined over output devices and shapes. Default `display` algorithms would be provided for an arbitrary shape on each 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, an `OutputDevice` class could introduce a `display` method:

```
public class OutputDevice {
    /* ... */
    public void display(Shape s) {
        Iterator pairs = orderedPairsOf(s.borderPoints());
        while (pairs.hasNext()) {
            renderLine( (Pair)pairs.next() );
        }
    }
    public native void renderLine( Pair p );
}
```

Each subclass of `OutputDevices` would be able to provide additional overriding `display` multimethods for particular kinds of shapes. For example, a `FastHardware` class might provide a few `display` multimethods:

```
public class FastHardware extends OutputDevice {
    /* ... */
```

```
    public void display(Shape s) {
        renderPolygon( s );
    }
    public void display(Shape@Rectangle r) {
        renderRectangle( r );
    }
    public void display(Shape@Circle c) {
        renderCircle( c );
    }
    public native void renderPolygon( Shape s );
    public native void renderRectangle( Rectangle r );
    public native void renderCircle( Circle c );
}
```

Augmenting methods added to open classes can also be multimethods. For example, the above `display` generic function could be implemented as an external generic function. (Perhaps mapping from the `Shape` representations to a representation supported by the `OutputDevice` instances.)

2.2.5. 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. We discuss this restriction further in Subsection 2.4.2.

2.3. Open Classes and Multimethods

One question that arises concerning MultiJava is why are open classes necessary given that the language includes multimethods. For example, one might attempt to add a generic function for calculating the area of shapes using multimethods instead of open classes:

```
public class AreaCalculator {
    public double areaFor( Shape s ) { /* ... */ }
    public double areaFor( Shape@Rectangle r ) { /* ... */ }
    public double areaFor( Shape@Circle c ) { /* ... */ }
}
```

With this definition a client can find the area of a shape as in the following:

```
double a = new AreaCalculator().areaFor( someShape );
```

There are two problems with this approach. First, the invocation syntax for the area-calculating generic function is different than for generic functions declared in regular Java or using open classes. The second problem is more onerous. As with the visitor pattern, we could no longer add new `Shape` subclasses in a modular way. New subclasses would require either a non-modular editing of the `AreaCalculator` class or an additional subclass of `AreaCalculator` with the associated problems as described on page 4 for the related subclassing solution to the augmenting method problem. (For

example, non-modular editing would be needed for client code that instantiated an `AreaCalculator` and was passed an instance of a new subclass of `Shape`. Otherwise the client would not have access to the algorithm defined for the new shape subclass in the new `AreaCaculator` subclass.)

Thus, multimethods alone do not provide a satisfactory solution to the augmenting method problem. Conversely, open classes provide no help with the binary method problem. Thus to solve both problems MultiJava includes open classes and multimethods.

2.3.1. Upcalls

Most single-dispatch object-oriented languages provide a mechanism whereby a method *M* may invoke a method of *M*'s immediate superclass; we call such an invocation a *superclass method invocation*. Java's `super` construct provides such a mechanism [Gosling et al. 2000] (§15.12).[16] A Java superclass method invocation may invoke a directly overridden method. Let M_2 be a method overridden by *M*. We say that *M directly overrides* M_2 if there exists no method M_3 such that *M* overrides M_3 and M_3 overrides M_2. Invoking a directly overridden method is the typical use for superclass method invocations; this idiom allows a method to "inherit" the behavior of an overridden method. In Java a superclass method invocation may also call a method in a different generic function than the sender, 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.

Unlike methods in Java, a MultiJava multimethod may override a method (or several) in the *same* class. For example, in Figure 10 on page 28 the third `overlaps` method of `Circle` overrides the first `overlaps` method in the same class. External methods may also override other methods in the same compilation unit. Thus, to take advantage of the inheritance of behavior from an overridden method, we want MultiJava to include a mechanism for invoking an overridden method with the same receiver type as the sending method. We call an invocation that targets a directly overridden method of the sender an *overridden method invocation*. This term applies whether or not the directly overridden method has the same receiver class as the sender. In other words a regular Java superclass method invocation to the same generic function as the sender is an overridden method invocation; a regular Java superclass method invocation to a different generic function is not. Overridden method invocations in MultiJava should be able to walk up a chain of overriding methods, even within the same class

16. We are concerned here with calls like `super.toString()`, not the use of `super()` for invoking a superclass constructor.

MethodInvocation:
```
...
overriddenMethod ( ArgumentList_opt )
```

Figure 11: Syntax extensions for MultiJava overridden method invocations: See Figure 5 on page 17 for a description of the notation.

or compilation unit. Regular Java superclass method invocations do not support this. Figure 11 gives the additions to the Java syntax for overridden method invocations in MultiJava.[17]

The term *upcalls* is used in this work to refer to both superclass method invocations and overridden method invocations. The remainder of this section discusses the semantics of superclass method invocations and overridden method invocations in MultiJava.[18]

Superclass Method Invocations

Superclass method invocations in MultiJava have the same semantics as in regular Java. That is, a superclass method invocation invokes the most specific applicable method of the target generic function whose receiver is a proper superclass of the sender. However, superclass method invocations pose an additional challenge for open classes that was not considered in CLCM2000. Arbitrary superclass method invocations from external methods can break the encapsulation of subclasses. Figure 12 demonstrates this. Part a) of the figure shows a portion of an `ActivityLog` class that might be used for recording transactions in an application. Part b) gives a subclass, `ProtectedLog`, that maintains a backup copy of the log file. Suppose a client is given an instance of `ProtectedLog`. There is no way in regular Java for the client to bypass the `close` method of `ProtectedLog` and directly call `ActivityLog`'s (non-backed-up) `close` method. In particular the message send

```
((ActivityLog) log).close(),
```

where `log` is an instance of `ProtectedLog`, always invokes `ProtectedLog`'s `close` method, because the explicit cast does not affect dynamic dispatch. However, with arbitrary superclass method invocations from external methods the client could call `ActivityLog`'s `close` method. To wit, by importing the generic function from Figure 12c, the client could bypass the `close` method of `ProtectedLog` using the message send `log.subvert()`.

Because superclass method invocations to different generic functions from within external methods can cause encapsulation problems, such invocations are disallowed. However, it is legal for an

17. In CLCM2000 we did not differentiate syntactically or in our terminology between superclass method invocations and overridden method invocations, using the term "super send" and the syntax `super.m()` for both. A static check differentiated between the two sorts of invocations and selected the appropriate semantics. This proved to be confusing and so we have separated the concepts and syntax.
18. As of this writing upcalls are not yet implemented in our compiler.

a)
```
public class ActivityLog {
    /* ... */
    public void close() {
        writePendingMessages();
        logFile.close();
    }
}
```

b)
```
public class ProtectedLog extends ActivityLog {
    /* ... */
    public void close() {
        super.close();
        logFile.copyTo( backupFile );
    }
}
```

c)
```
// compilation unit "subvert"
public void ActivityLog.subvert() { this.close(); }
public void ProtectedLog.subvert() { super.close(); }// breaks encapsulation
```

Figure 12: Encapsulation problem with superclass method invocations from external methods: part a) shows a Java class for maintaining a log of activity in some application, part b) gives a subclass that maintains a backup copy of the log file, part c) shows an external generic function that could break the encapsulation of `ProtectedLog`

internal method of an external generic function to invoke a superclass method invocation on a different generic function. Such invocations do not cause encapsulation problems and in any case must be supported for compatibility with Java.

Overridden Method Invocations

In MultiJava, the `overriddenMethod(arg1, ..., argn)` expression allows a method to invoke the method it directly overrides, whether or not the target method has the same receiver class as the sender. The target of an overridden method invocation must be a unique, statically-determined method body.[19]

Consider an implementation of the third `overlaps` method of `Circle` (from Figure 10 on page 28) that contains an overridden method invocation:

19. With multimethods it is possible for a single method to directly override more than one other method. An invocation of `overriddenMethod()` in such a method would be ambiguous and is statically rejected in MultiJava. Subsection 6.1.1 on page 91 gives a syntax-extension and implementation strategy that would allow a programmer to resolve such ambiguities by specifying which of the directly overridden methods is to be invoked.

```
public boolean overlaps(Shape@Circle c) {
    ... overriddenMethod(c) ...
}
```

In this case the directly overridden method is the `overlaps(Shape s)` method declared within the `Circle` class and so it will be invoked. If that method itself contains an overridden method invocation of the same form, then the directly overridden method would the one declared within the `Shape` class. In this case the receiver of the target method is different than the receiver of the sender. An overridden method invocation in `Shape`'s `overlaps` method would lead to a static type error, as there would be no applicable methods.

Because of the restrictions on the location of augmenting method declarations (introduced in Subsection 2.1.6 and elaborated on in "Unrestricted Method Overriding" on page 41), an overridden method invocation from within an external method will always invoke a method declared in the same compilation unit. Thus such overridden method invocations cannot cause the encapsulation problems discussed for superclass method invocations.

2.4. Typechecking MultiJava

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 2.4.1. In Subsection 2.4.2 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.

2.4.1. 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 and Leavens 1995]. We begin by reiterating what we mean by modular typechecking, particularly in the context of method invocations, and then discuss the two kinds of checks.

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 an invocation generating a message-not-understood or message-ambiguous error at run time.

We say that a type is *directly 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 type is *visible* in a compilation unit *U* if it is directly visible in *U* or if (recursively) it is visible in a type that is directly visible in *U*. 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 generic function that is imported by *U*. A generic function is *visible* in a compilation unit *U* if any of its methods are visible in *U*. A modular typechecking strategy only needs to consider visible types and visible methods to determine whether a compilation unit is type-correct.

Client-side Typechecking

Client-side checks are local checks for type correctness of each message send expression. In general these checks just extend those for regular Java [Gosling et al. 2000] (§15.12.1–3).

Briefly, 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, most specific visible generic function named *I* whose top method has a tuple of argument types $(T_0', \ldots, T_1')$ that is a supertype of $(T_0, \ldots, T_1)$. It is possible for the target generic function to be statically ambiguous, even in regular Java. For example, given the following declaration:

```
public class ComplexNumber {
    /* ... */
    public ComplexNumber add( Real r ) {
        return new ComplexNumber( real().add(r), complex() );
    }
    public ComplexNumber add( Imaginary i ) {
        return new ComplexNumber( real(), complex().add(i) );
    }
}
```

the invocation `new ComplexNumber().add(null)` is ambiguous between the two statically overloaded generic functions.

Once the target generic function is determined the return type of the generic function and the possible exceptions thrown are calculated exactly as in regular Java.[20] In our extension, however, external generic functions that are imported must be checked along with regular class and interface declarations.

For a superclass method invocation (i.e., 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.

20. Note that the exceptions thrown are determined based on those for the method of the generic function declared in the nearest (possibly reflexive) supertype of the static type of the invocation's receiver. These exceptions may be a subset of those declared for the generic function's top method.

Such a method would necessarily be declared in a superclass of the calling method's receiver class or in an external generic function augmenting such a superclass. This check extends (to consider external generic functions) the checking on superclass method invocations that Java performs already. Also, as discussed in Subsection 2.3.1, to preserve encapsulation the checks prevent superclass method invocations to different generic functions from within external methods. For overridden method invocations, the checks verify the existence of a unique, directly-overridden method for the calling method. For both kinds of upcalls the return type and exceptions thrown by the target method are subsequently used for typechecking the expression containing the upcall.

Implementation-side Typechecking

Implementation-side checks ensure that each generic function is fully and unambiguously implemented. These checks can be grouped into two sets.

CHECKS ON INDIVIDUAL METHOD DECLARATIONS. The first set of checks applies to each method declaration *M* in isolation. The first check of this set is on the explicit specializers:

> For each of *M*'s specialized parameter types, *S*`@`*D*, *S* must be a reference type, *S* must be a proper supertype of *D*, and *D* must be a class type or class array type.

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 or array types, rather than interfaces, because the form of multiple inheritance supported by interfaces can create ambiguities that elude modular static detection [Millstein and Chambers 1999].[21] Although currently prohibited, it seems possible to allow `null` as an explicit specializer. Subsection 6.1.6 on page 96 briefly discusses the changes to the typechecking and implementation strategies needed to support this.

The remainder of the individual method declaration checks compare a method declaration *M* against the declarations of each of *M*'s directly overridden methods. Several of these checks are identical to the checks already performed in Java [Gosling et al. 2000] (§8.4). It might seem that these checks must be more complex in MultiJava since a single multimethod can directly override several

21. This does not preclude using an interface as the static type in a multimethod with a class type as the explicit specializer.

other methods. However, a single pleomorphic method in regular Java can also directly override several other methods so this complication must already be considered.

The checks that are unchanged from regular Java include verifying that

- no directly overridden method is declared `final`,
- *M* has the same return type as each of *M*'s directly overridden methods,
- the exceptions declared by *M* are compatible with those of *M*'s directly overridden methods, and
- *M*'s privileged access level is no more restrictive than that of any of *M*'s directly overridden methods.

Besides these checks that are unchanged from regular Java, MultiJava includes additional overridden method checks that apply only to multimethods or methods of external generic functions. These checks are strictly more restrictive than those for regular Java and are intended to simplify the compilation scheme.

Section 3 on page 45 discusses the compilation strategy in detail, but to understand the individual checks on multimethods and methods of external generic functions it is helpful to provide a brief preview here. A key tactic of the compilation strategy is to create a unique *dispatcher method* that houses all the methods of a single generic function that appear in a single context (i.e., a single class for internal methods or a single compilation unit for external methods). The following checks ensure that the methods housed in a dispatcher method are compatible. The checks are further complicated for external generic functions, where the dispatcher methods will form a linked list and all methods of the generic function must be compatible.

If a method *M* belongs to an internal generic function, then we refer to all the other methods housed in the same dispatcher method as the *dispatcher mates* of *M*; if *M* belongs to an external generic function, then we refer to all the other methods in the entire generic function, as the dispatcher mates of *M*. For our compilation strategy we verify that for every dispatcher mate of *M:*

- *M* has the same modifiers, including privileged access level, and
- *M* declares the same exceptions.

If *M* is an internal method without explicit specializers and belongs to an internal generic function, i.e., *M* is a regular Java method, then *M* has no dispatcher mates. Thus the dispatcher mate restrictions are vacuously true and only the regular Java restrictions for overriding methods apply. In

Subsection 6.1 we briefly investigate ways of relaxing these compilation-strategy induced restrictions so that overriding multimethods and methods of external generic functions may be checked using the regular Java restrictions only.

In practice these dispatcher mate checks on *M* only need to be applied to a subset of *M*'s dispatcher mates, called "checkmates$_M$". Define a set *checkmates$_M$* as follows:

- If *M* is an internal method then *checkmates$_M$* is the set of all methods in the class containing *M* that are directly overridden by *M*, that is, the dispatcher mates of *M*.
- Otherwise, if *M* is an external method then *checkmates$_M$* is just the singleton set containing the top method of *M*'s generic function. Since each method in the generic function shares the same top method, checking each method against this singleton set suffices to verify that the above conditions hold for all dispatcher mates of *M*.

CHECKS ON ENTIRE GENERIC FUNCTIONS. The second set of 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 $(T_0, \ldots, T_1)$. A tuple of types $(C_0, \ldots, C_1)$ is a *legal argument tuple* of the generic function if $(T_0, \ldots, T_1)$ is a supertype of $(C_0, \ldots, C_1)$ and each C_i is concrete. We say that a type is *concrete* if it is a primitive type, if it is a class that is not declared `abstract`, or if it is an array type. Interfaces and abstract classes are *non-concrete*. Conceptually the set of visible legal argument tuples represents all the possible combinations of arguments that might occur at run-time. The checks are that for each visible generic function to which a local method belongs, each visible legal argument tuple has a visible, most-specific applicable method to invoke. A method is *local* if it is declared in the compilation unit being checked. 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 `overlaps` generic function, from the perspective of a compilation unit containing only the `Rectangle` class as defined in Figure 9. From this compilation unit, `Shape` and `Rectangle` are the only visible `Shape` subclasses (`Circle` and `Parallelogram` are not visible, because they are not referenced by the `Rectangle` class). The `overlaps` generic function is visible, as are two `overlaps` methods (one each in `Shape` and `Rectangle`). There are four visible legal argument tuples: all pairs of `Shapes` and `Rectangles`. The `overlaps` method in class `Rectangle` is the most specific applicable method for the (`Rectangle`, `Rectangle`) tuple while the `overlaps` method in class `Shape` is the most specific applicable method for the other three tuples.

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

Figure 13: Incompleteness problem with abstract classes and open classes

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 and Leavens 1995, Castagna 1997].

2.4.2. 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 [Cook 1991, Chambers and Leavens 1995, Millstein and Chambers 1999]. This is the modularity problem that was introduced in Subsection 1.1.3. We say a generic function is *incomplete* if it can cause message-not-understood errors at run time. We say a generic function is *ambiguous* if it can cause message-ambiguous errors at run time. In the rest of this subsection, we describe the ways that these errors can occur, and explain the restrictions we impose in MultiJava to rule them out.

Abstract Classes and Open Classes

As mentioned in Subsection 2.1.6 on page 23, abstract external methods can lead to message-not-understood errors. This is illustrated in Figure 13. 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***:

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

Figure 14: Incompleteness problem with abstract classes and multimethods

(***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.

As with methods, a type 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 13, 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 `Pictures`, which resolves the incompleteness for the unseen `JPEG` class.

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 causing static typechecking to signal an incompleteness error for the generic function on that receiver class. For the same reason, MultiJava cannot support open interfaces, i.e., the ability to add method signatures to interfaces.

Abstract Classes and Multimethods

Abstract classes coupled with multimethods can also lead to message-not-understood errors. Consider the example in Figure 14. 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 14, 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 14 the abstract Picture class may safely omit an implementation of the internal similar generic function.

Unrestricted Method Overriding

Message-ambiguous errors that elude static detection can occur if methods can be arbitrarily added to a generic function by any compilation unit. These errors can occur without multiple dispatch (as mentioned in Subsection 2.1.6 on page 23). In this section we give an example that uses multiple dispatch.

Consider the example in Figure 15, assuming the Shape class from Figure 2 on page 9 and the Rectangle class from Figure 9 on page 25. The external method declaration in compilation unit *overlaps* overrides the default Shape overlap-detection method for argument tuples with dynamic type (Shape, Rectangle). Shapes and Rectangles are visible in the *overlaps* compilation unit, and every pair of these classes has a most-specific applicable method. Similarly, Shapes and Triangles are visi-

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

Figure 15: Ambiguity problem with unrestricted multimethods

ble in the *Triangle* compilation unit of Figure 15, though `Rectangle`s are not, so *Triangle*'s implementation-side checks also succeed. However, for a client that imports both `overlaps` and `Triangle`, an `overlaps` message send with type tuple (`Triangle`, `Rectangle`) will cause a message-ambiguous error to occur at run-time, 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 Jr. 1990, Paepcke 1993] and Polyglot [Agrawal et al. 1991], then the method in `Triangle` would be strictly more specific than the method in *overlaps*. 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 either be the top method of a new generic function or must override only local methods.

In Figure 15, the external method declaration in the *overlaps* compilation unit violates restriction ***R3***. In particular, the associated `overlaps` method overrides a method in the non-local `Shape` class. By restriction ***R3***, the only legal location for the declaration of an `overlaps` 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.

A corollary of restriction ***R3*** is:

```
// compilation unit "force"
public void Point.force(Point p) { ... }
public void Electron.force(Point@Electron e) { ... }// disallowed by R3

// compilation unit "Point"
public class Point { ... }

// compilation unit "Charge"
public class Charge extends Point {
    public void force(Point@Charge c) { ... }
}

// compilation unit "Electron"
public class Electron extends Charge { ... }
```

Figure 16: Code permitted under restriction ***R3-relaxed***

(***R3a***) If a method *M* overrides some non-local method, then *M* must be internal.

Each method declaration *M* must be in the same compilation unit as either the receiver's class (by ***R3a***) or the associated generic function's top method (by transitivity on ***R3***). 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.

The CLCM2000 definition of MultiJava used a more relaxed version of this restriction:

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

This original restriction permitted code like that shown in Figure 16.[22] In the figure the external method `Electron.force(Point@Electron e)` belongs to the local generic function with top method `Point.force(Point p)` and so was permitted by restriction ***R3-relaxed***. However, `Electron.force(Point@Electron e)` overrides the non-local method `Charge.force(Point@Charge c)` declared in compilation unit *Charge*. Thus this code is disallowed by restriction ***R3***. The MultiJava type system using ***R3-relaxed*** is sound; the tighter restriction presented here is not necessary for type-safety. Rather the version of ***R3*** presented here is intended as a software engineering restriction that improves the readability of code. For a programmer to understand the methods overridden by an internal method he or she only needs to consider the local class, superclass, and implemented interfaces (as

22. This example is due to Jason Baker (personal communication).

in regular Java), along with any imported generic functions. To understand the methods overridden by an external method the programmer only needs to consider the local compilation unit, as opposed to looking in the superclasses of all receiver classes. For the same reason, ***R3*** also simplifies compilation by allowing the compiler to combine all external methods of a single generic function into a single bytecode method.

Summary of Restrictions

These restrictions are necessary for modular, static typechecking and compilation. They still permit many common coding patterns, including those necessary to solve the augmenting and binary method problems.

While we can solve the binary problem in its usual definition, there is a generalized version of the binary method problem that is beyond the ability of this version of MultiJava to solve with modular editing. In the usual definition of the binary method problem it is sufficient that the solution allows one to write methods that dispatch on pairs of objects of the same type. Clearly MultiJava supports this, including modular addition of such methods as new subclasses are added.

In the *generalized binary method problem* we wish to write methods that dispatch on all possible pairs of objects from a given class hierarchy. Suppose we have an existing class hierarchy with multimethods of some internal generic function implemented for all pairs of classes in the hierarchy. If we add a class *C* to the hierarchy it is a simple matter to implement multimethods in *C* that handle all pairs of classes in which *C* is the first element. However, under the current restrictions a non-modular editing of the original classes is necessary to handle pairs of classes where *C* is the second element. The restrictions necessary for modular, static typechecking and compilation do not provide enough expressiveness for solving the generalized binary method problem. (The situation in MultiJava is still better than double-dispatching where non-modular editing is needed of the root class even to handle the pairs where *C* is the first element.)

A type system based on System E of the Dubious language [Millstein and Chambers 1999] would allow one to write methods for pairs where *C* is the second element. System E does not provide fully modular typechecking; link-time checks are required to prevent the sorts of ambiguity and incompleteness described above. Subsection 6.1.7 on page 97 discusses this in more detail.

SECTION 3. CODE GENERATION STRATEGY

The compilation strategy for MultiJava generates standard Java bytecode and retains the modular compilation and efficient single dispatch of existing Java code while supporting the new features of open classes and multiple dispatch. Additional run-time cost for these new features is incurred only where such features are used; code that does not make use of multiple dispatch 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. A method of 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 Java method. A method of 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 (perhaps transitively) both the compilation unit declaring the generic function’s top method and the one declaring the top method’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 3.2 describes the same for external generic functions. Subsection 3.3 and Subsection 3.4 describe the compilation of pleomorphic methods and upcalls respectively. Finally, Subsection 3.5 discusses some miscellaneous compilation issues. Although the compilation outputs Java bytecode, to simplify discussion we will generally describe compilation as if going to Java source code. However, in some situations we need to exploit the additional flexibility of compiling directly to the Java virtual machine.

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

b)
```
public class Square extends Rectangle {
    /* ... */
    // the "overlaps" dispatcher method
    public boolean overlaps(Shape r) {
        if (r instanceof Square) {
            return overlaps$body((Square) r);
        } else if (r instanceof Rectangle) {
            return overlaps$body((Rectangle) r);
        } else {
            return super.overlaps(r);
        }
    }
    private boolean overlaps$body(Rectangle r) {
        /* method 1 body */
    }
    private boolean overlaps$body(Square s) {
        /* method 2 body */
    }
}
```

Figure 17: Internal generic function and its translation:
part a) shows two internal methods of an internal generic function,
part b) shows their translation into regular Java code.

3.1. Internal Generic Functions

All the multimethods of an internal generic function with the same receiver class are accessed as a unit via a single Java method that we call a *dispatcher method*. Consider the set of `overlaps` methods in Figure 17a. For such a set of multimethods, the MultiJava compiler produces a dispatcher method within the receiver class that dispatches to the body of the appropriate multimethod in the set. The multimethod bodies are translated into a set of overloaded private methods. Figure 17b shows the result of translating the MultiJava code from Figure 17a. In the translation, the dispatcher method has the same name as the generic function (`overlaps` in this case), and has the same static argument types as all the generic function's methods. The dispatcher 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. The multimethod bodies are translated into private methods whose names are the concatenation of the generic function name with the suffix `$body` (`overlaps$body` in this case).[23] By using private methods for multimethod bodies, instead of inlining the code, we avoid code duplication when multiple paths through these `instanceof` sequences lead to the same method body. Alternatively, `goto` instructions could be exploited to allow inlining of most code while avoiding code duplication. In lieu of cascaded sequences of `instanceof` tests, there are other efficient dispatching schemes that could be exploited [Chambers and Chen 1999].

For the set of multimethods compiled into a dispatcher 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, because the receiver position is more specific for the subclass, so there is no need to check superclass multimethods before dispatching to a local multimethod.

If every multimethod compiled into a dispatcher 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 single dispatch 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. Of course in this case the method body is not relocated to a separate private method.

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

---

23. Unique integers could be used (as in `overlaps$1$body`) to avoid name clashes in the presence of statically overloaded generic functions. As of this writing our compiler does not implement this.

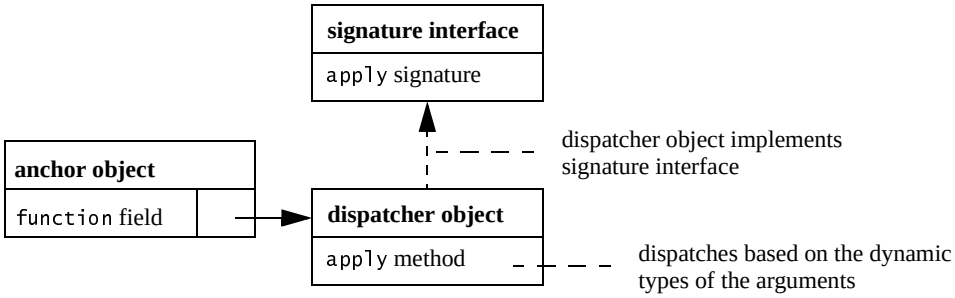

Figure 18: Objects used in the compilation of external generic functions

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 the only performance difference should come from the dispatch to the private method representing the multimethod body. As already noted, this penalty could be eliminated by inlining the multimethod bodies. An efficient JVM implementation can also dynamically inline these private methods [Gosling et al. 2000] (see §8.4.3.3 and note that private methods are implicitly final). 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. Section 5 on page 69 presents some empirical data on dispatch performance.

3.2. External Generic Functions

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

Figure 18 shows the objects generated in the compilation of an external generic function. An anchor class instance has a single static field, `function`, containing a *dispatcher object*. The dispatcher object implements a *signature interface* that is used for extending external generic functions (see below).[24] 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. The dis-

24. For overloaded external generic functions the anchor class contains multiple function fields, one for each generic function, and multiple signature interfaces are created.

a)
```
// 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 */ }
```

b)
```
public class rotate$anchor {// an anchor class
    public interface signature {   // type of a dispatcher object in this example
        Shape apply(float a, Shape this_);
    }
    public static rotate$anchor.signature function =
        new dispatcher();
    // a nested class implementing a dispatcher object
    private static class dispatcher implements rotate$anchor.signature {
        public Shape apply(float a, Shape this_) {
            if (this_ instanceof Square) {
                return rotate$body(a, (Square) this_);
            } else if (this_ instanceof Rectangle) {
                return rotate$body(a, (Rectangle) this_);
            } else {
                return rotate$body(a, this_);
            }
        }
        private static Shape rotate$body(float a, Shape this_) {
            /* method 3 body, substituting this_ for this */
        }
        private static Shape rotate$body(float a, Rectangle this_) {
            /* method 4 body, substituting this_ for this */
        }
        private static Shape rotate$body(float a, Square this_) {
            /* method 5 body, substituting this_ for this */
        }
    }
}
```

Figure 19: External generic function and its translation:
part a) shows three external methods declaring a new external generic function,
part b) shows their translation into regular Java code

patcher object contains all the methods of a particular generic function that are declared in a single compilation unit. It is a Java version of a first-class function, allowing the generic function's methods to be stored in a field.

As an example, Figure 19a introduces the `rotate` external generic function and its first three methods. Figure 19b shows the results of compiling it. The privileged access level of the top method determines the privileged access level of the anchor class and its `function` field. The name for the

anchor class is formed by concatenating the generic function name with the suffix `$anchor`. Thus in this example, the anchor class is named `rotate$anchor`. The signature interface is a nested interface, named `signature`, within the anchor class. As with internal generic functions, dispatching is performed using cascaded `instanceof` tests; the same optimizations apply. Since the methods do not appear in the same class as their logical receivers, the "receiver" of the message send is passed as an extra argument.

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.overlaps(s2)) {
    s2 = s2.rotate(90.0);
}
```

is translated to:

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

As with internal generic functions, clients invoking external generic functions are insensitive to whether or not the generic function performs any multiple dispatch. 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 ***R3a***. 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 20 shows a new dispatcher object created for such a set of methods. The anchor class's `function` field from Figure 18 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 (forming a linked list). 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 21a shows a class, `Oval`, containing a method that is added to the non-local external generic function, `rotate`. Figure 21b shows the results of compiling this class. A new nested

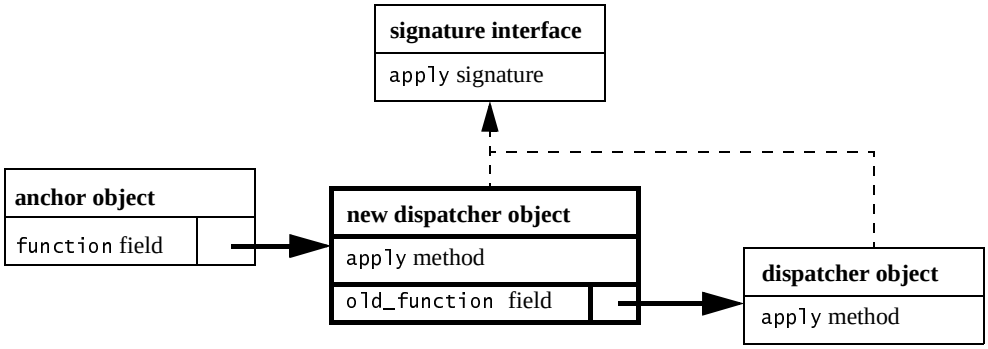

Figure 20: Objects used when adding methods to non-local external generic functions

dispatcher class, `Oval.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 anchor class's `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. 1995] (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; Section 5 gives some empirical results. Subsection 3.5 includes a discussion of two strategies that could be employed to improve the efficiency of dispatch to external generic functions, one involving dynamic compilation and the other using a customized JVM.

The order in which dispatcher objects are checked depends on the order in which they are put into the chain referenced by `rotate$anchor`'s `function` field. Java ensures that superclasses are initialized before subclasses [Gosling et al. 2000] (§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. Two unrelated classes might have their dispatchers put onto the

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

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

Figure 21: Internal method of an external generic function and its translation:
part a) shows an internal method that overrides methods of an external generic function,
part b) shows its translation into regular Java code

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 2.1, 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 a nested class in the corresponding receiver class [Gosling et al. 2000] (§6.6.2).

3.3. Pleomorphic Methods

In Subsection 2.1.2 on page 18 we noted that a method can be pleomorphic—it can simultaneously belong to more than one generic function. Because of Java's single inheritance and the ambiguity detection of the MultiJava type system, only one of the generic functions to which a pleomorphic method belongs will have a concrete top method. The other generic functions must be declared in interfaces.

It is possible that only one of the generic functions to which a pleomorphic method belongs is visible to a particular client. This is not a problem in regular Java. If the client-visible generic function is declared in an interface, then an `invokeinterface` instruction will be used for the message send; if the client-visible generic function is declared in a class, then an `invokespecial` instruction will be used. But both instructions can resolve to the same pleomorphic method at run-time. Figure 22 gives an example. The method `ListSet.contains()` in part a) is pleomorphic, belonging the generic function declared by the interface `Set` and the one declared by the class `List`. In part b) the invocation `s.contains(o)` will be compiled into an `invokeinterface` instruction, since only the `Set.contains()` generic function, declared in an interface, is visible. On the other hand in part c) the invocation `l.contains(o)` will be compiled into an `invokevirtual` instruction, since only the `List.contains()` generic function, declared in a class, is visible. At run-time if an instance of `ListSet` is passed to the `process()` method of either client, then the body of the `ListSet.contains()` method will be executed.

The challenge that arises in MultiJava is that a method may be pleomorphic on an internal and an external generic function. Very different calling conventions are used for the two kinds of generic function. And, based on the compilation strategy described thus far, the body of the pleomorphic method must appear in both a nested class in the external generic function's chain of responsibility and inside the receiver class. Our strategy is to compile the pleomorphic method according to the strategy for external generic functions. But within the receiver class we create a *redirector method* that directs an invocation of the internal generic function on that class into the external generic function.

For example, notice that the `Set` interface of Figure 22, part a) declares a method `size` and `ListSet` implements that method. Now suppose there exists a declaration for an external generic function `size` as in Figure 23, part a). Since this declaration is in the same package as the declaration of `ListSet`, the `size` method of `ListSet` is pleomorphic between the internal generic function whose top method is declared in `Set` and the external generic function whose top method is declared in *size*. An

a)
```
// compilation unit "Set"
package thesis.types;
public interface Set {
    /* ... */
    boolean contains(Object o);
    int size();
}
```

```
// compilation unit "List"
package thesis.containers;
public class List {
    /* ... */
    boolean contains(Object o) { /* ... */ }
}
```

```
// compilation unit "ListSet"
package thesis.moreContainers.containers;
import thesis.containers.List;
import thesis.types.Set;
public class ListSet extends List implements Set {
    /* ... */

    // pleomorphic method, belongs to Set.contains() and List.contains() generic functions
    boolean contains(Object o) { /* method 7 body */ }

    int size() { /* method 8 body */ }
}
```

b)
```
// compilation unit "ClientOne"
package thesis.examples;
import thesis.types.Set;
public class ClientOne {
    /* ... */
    public void process(Set s) {
        ... s.contains(o);
        ...
    }
}
```

c)
```
// compilation unit "ClientTwo"
package thesis.examples;
import thesis.containers.List;
import thesis.moreContainers.size;
public class ClientTwo {
    /* ... */
    public void process(List l) {
        ... l.contains(o);
        ...
    }
}
```

Figure 22: Example showing a pleomorphic method in regular Java code:
part a) introduces two compilation units, each declaring a new generic function, and a class `ListSet` declaring a pleomorphic method, part b) gives a client for which only the `Set.contains` generic function is visible, part c) gives a client for which only the `List.contains` generic function is visible.

a)
```
// compilation unit "size"
package thesis.moreContainers;

import thesis.containers.List;

public int List.size() {
    return elements().length;
}
```

---

b)
```
package thesis.moreContainers;
import thesis.containers.List;
import thesis.containers.Set;
public class ListSet extends List implements Set{

    // static initializer:
    static {
        /* ... */
        size$anchor.function =
            new dispatcher(size$anchor.function);
    }

    /* ... */

    // redirector method for the pleomorphic method
    public int size() {
        return contains$anchor.function.apply(this);
    }

    // a nested class implementing a dispatcher object for the external generic function
    private static class dispatcher implements size$anchor.signature {
        public size$anchor.signature oldFunction;
        public dispatcher(size$anchor.signature oldF) {
            oldFunction = oldF;
        }
        public boolean apply(List this_) {
            if (this_ instanceof ListSet) {
                return contains$body((ListSet) this_);
            } else {
                return oldFunction.apply(this_);
            }
        }

        private static boolean contains$body(ListSet this_) {
            /* method 8 body, substituting this_ for this */
        }
    }

}
```

Figure 23: Additions to Figure 22 to demonstrate redirector methods: part a) gives an external generic function to which the `ListSet.size` method of Figure 22 belongs, part b) gives a part of the translation of the `ListSet` class that demonstrates the creation of redirector methods for pleomorphic methods.

invocation of `size` from within `ClientOne` of Figure 22, part b) would target the internal generic function and be compiled into an `invokeinterface` bytecode as described above. However, an invocation of `size` from within `ClientTwo` of Figure 22, part c) would target the external generic function and be compiled into an invocation of the `apply` method of the dispatcher object stored in the anchor class's `function` field. A redirector method is needed to divert the internal generic function invocation into the external generic function implementation.[25] Figure 23, part b) shows the code generated by the compiler for the ListSet class that is pertinent to the `size` generic function. Bold print in the figure indicates the redirector method for the pleomorphic method.

3.4. Upcalls

The compilation of superclass method invocations and overridden method invocations presents interesting challenges, which were not appropriately addressed in CLCM2000.[26] Because of the various compilation tactics for method definitions the compiled superclass method invocation or overridden method invocation may originate in a nested class of an anchor class (for external methods of an external generic function), a nested class of a regular Java class (for internal methods of an external generic function), or in a regular Java class (for internal methods of an internal generic function). The target method of the invocation may appear in the same variety of locations. Additionally, overridden method invocations, must target particular multimethods, which may or may not appear in the same bytecode class as the sender.

Thus there are a number of permutations of caller and target method locations for superclass method invocations and overridden method invocations. The compilation tactic used varies based on these permutations and between the two sorts of invocations. We consider superclass method invocations and overridden method invocations separately.

3.4.1. Superclass Method Invocations

Figures 24 gives a diagram illustrating the possible calling and target method locations for superclass method invocations in MultiJava, considering the restrictions for encapsulation discussed in Subsection 2.3.1 on page 31. Arrows in the figures are drawn from the calling method to the target of an invocation. Labels on the arrows correspond to the cases in the description below. (The class `A` at the

25. Another implementation of pleomorphic methods might duplicate the body of a pleomorphic method in both internal and external implementations.
26. As of this writing, the implementation of these features in `mjc` is not finished.

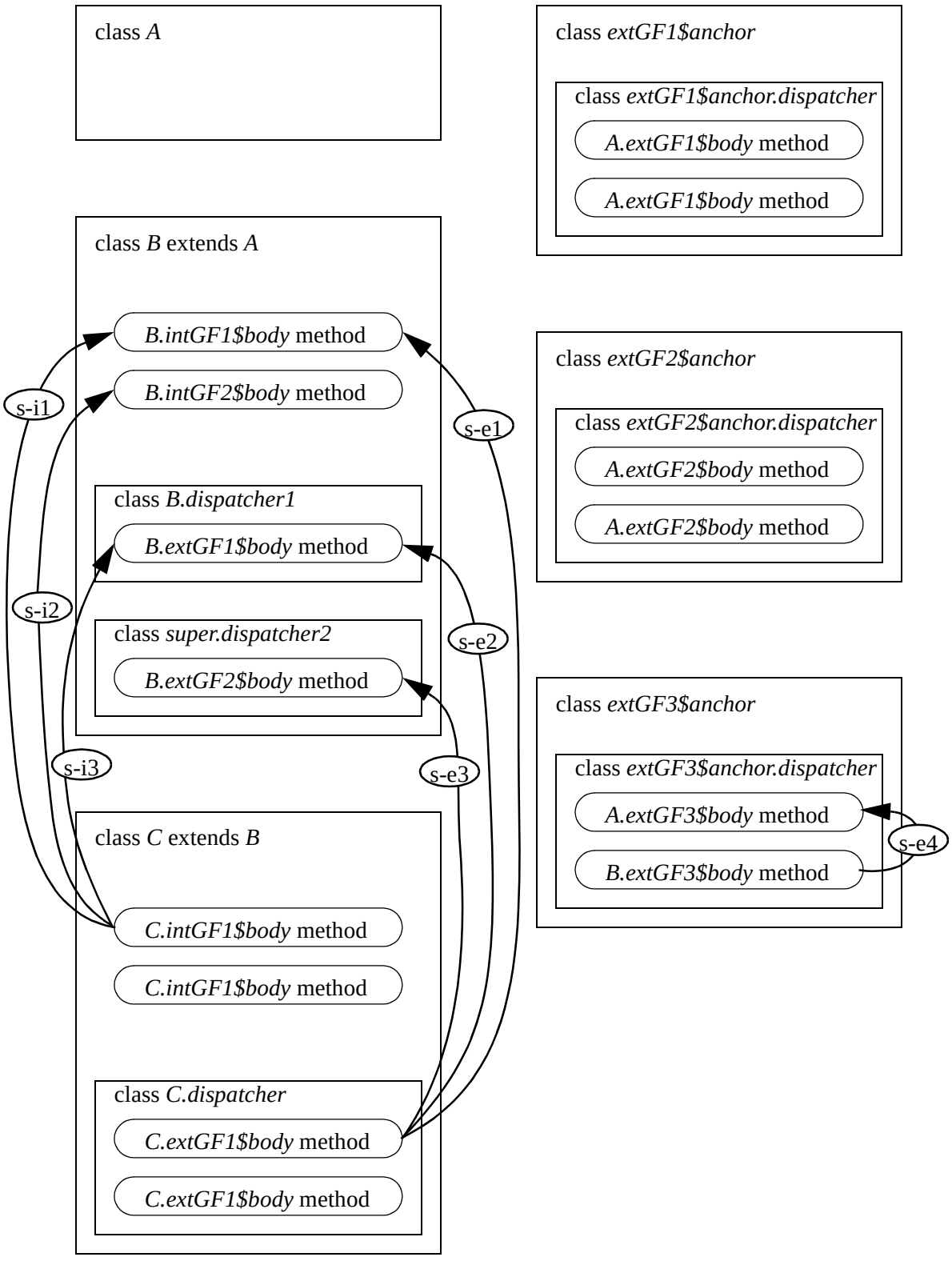

Figure 24: Legal combinations of target and sender locations for a superclass method invocation

top-left and the two anchor classes, at top-right and right-center, are those necessary for the methods participating in the superclass method invocations to appear in the indicated locations.)

As shown in the figure, there are three possible target locations for a superclass method invocation originating in an internal generic function. When the calling generic function is internal, and:

s-i1: the target generic function is the same as the calling generic function, then the invocation is just a regular Java `super`. It is compiled in the usual fashion using an `invokespecial` bytecode.

s-i2: the target generic function is an internal generic function different than the calling generic function, then as in case ***s-i1*** the call is just a regular Java `super` and the `invokespecial` bytecode is used.

s-i3: the target generic function is external, then there are two possibilities. Because this is a superclass method invocation, the target method's receiver class must be a proper superclass of the calling method's receiver class, *C*. If *C* does not add methods to the target generic function, then the invocation is compiled as a regular external generic function invocation. Because of the prohibition on self-augmenting classes and restriction ***R3***, we know that no method with receiver class *C* will be invoked. On the other hand, if *C* adds methods to the target generic function (as in the figure), then again the invocation is compiled like a regular external generic function invocation, but instead of using the dispatcher object from the `function` field of the anchor class, we use the value stored in the local dispatcher object's `old_function` field. This allows the superclass method invocation to bypass the local methods of the target generic function.

Figure 24 also shows the three possible target locations for a superclass method invocation originating in an internal method of an external generic function. When the calling method is an internal method of an external generic function, and

s-e1: the target generic function is internal, then we would like to treat this as a regular Java superclass method invocation using `invokespecial`. But the calling method is compiled in a nested class of *C*. The invocation is accomplished by adding a private method to *C* which is invoked by the calling method and which redirects to the appropriate method of *B* via an `invokespecial` bytecode.

s-e2: the target generic function is the same as the calling generic function, then the invocation is just a call to the next dispatcher object in the chain of responsibility, compiled using the `old_function` field as in case ***s-i3***.

s-e3: the target generic function is external and different than the calling generic function, then there are two possibilities. If *C* does not add methods to the target generic function (as in the figure), then the superclass method invocation is compiled as a regular external generic function invocation. As in case ***s-i3***, we know that no method with receiver class *C* will be invoked. On the other hand, if *C* adds methods to the target generic function, then again the superclass method invocation is compiled like a regular external generic function invocation, but using the value stored in the `old_function` field of the target generic function's local dispatcher object.

Finally, when the calling method is an external method of an external generic function,

s-e4: then, by the encapsulation restrictions of Subsection 2.3.1 and ***R3***, the target method must be in the same dispatcher object. The superclass method invocation is compiled as an invocation of separate dispatcher method that only dispatches to multimethod bodies whose receiver class is a proper supertype of the caller's.

3.4.2. Overridden Method Invocations

Figure 25 gives a diagram illustrating the possible calling and target method locations for overridden method invocations in MultiJava. Because only methods of the same generic function as the caller's can be targeted the number of cases here is less than in Figure 24. However, the analysis is similar. When the calling generic function is internal, and

o-i1: the target method is in the same class as the calling method, then the overridden method invocation is just a call to the appropriate *ident*`$body` method of the local class.

o-i2: the target method is a superclass, then as in case ***s-i1*** the call is just a regular Java `super` and the `invokespecial` bytecode is used.

When the calling method is an internal method of an external generic function, and

o-e1: the target method is declared in the same class as the calling method, then both caller and target are compiled in the same dispatcher object. The overridden method invocation is just a call to the appropriate *ident*`$body` method of that dispatcher object.

o-e2: the target method is not declared in the same class as the calling method, then the overridden method invocation is compiled like a regular external generic function invocation, but using the value stored in the local dispatcher object's `old_function` field. This allows the invocation to bypass the local methods of the target generic function.

Finally, when the calling method is an external method of an external generic function,

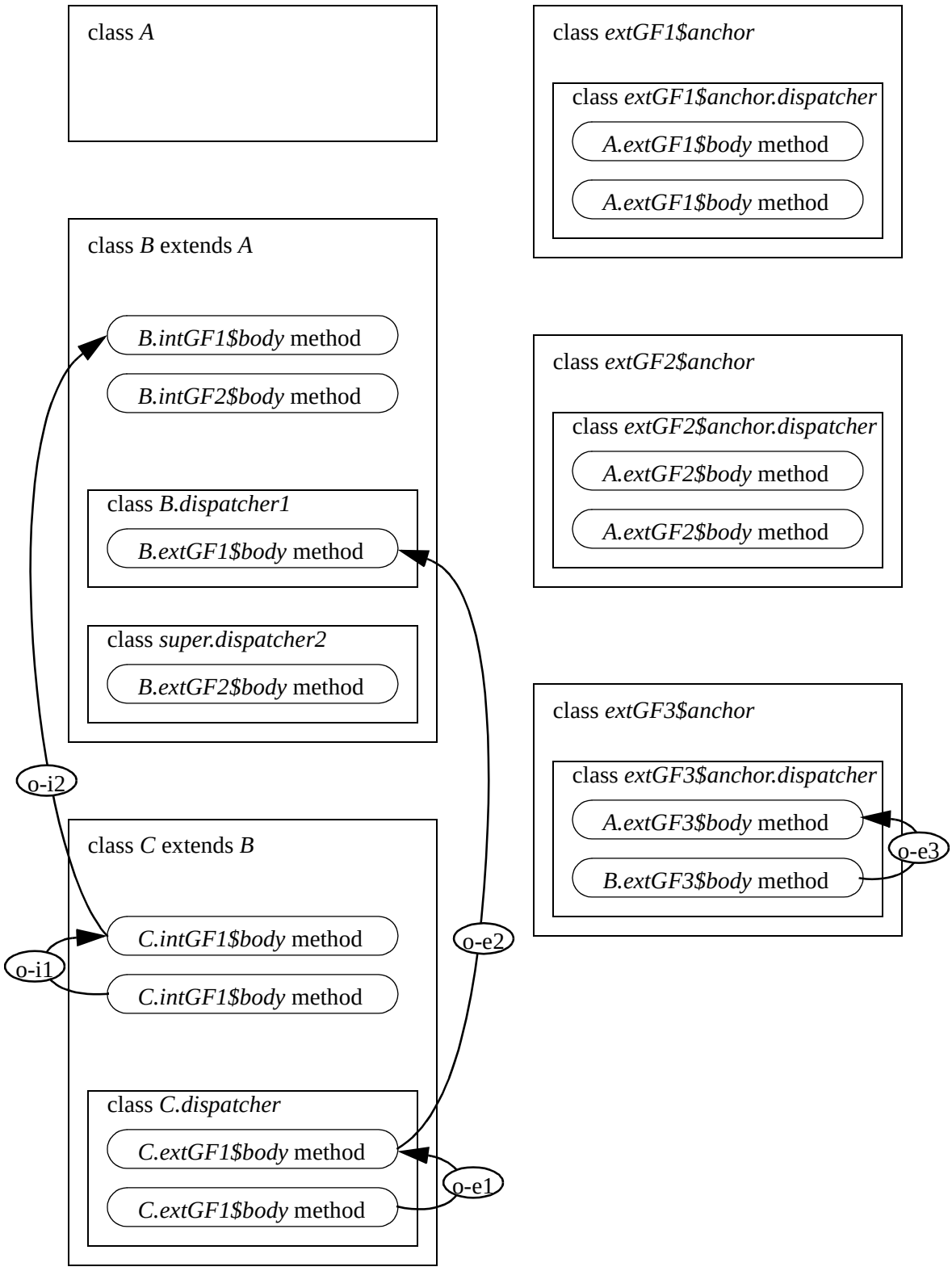

Figure 25: Legal combinations of target and sender method locations for an overridden method invocation

```
// compilation unit "sort"
package thesis.examples;
public void List.sort() {
    ... this.swap(i,j);
    ...
}
// private external helper method
private void List.swap(int i, int j) {
    Object temp = get(i);
    set(i, get(j));
    set(j, temp);
}
```

Figure 26: Compilation unit with a private external method

o-e3: then by R3, both caller and target are compiled in the same dispatcher object. The overridden method invocation is just a call to the appropriate *ident*`$body` method of that dispatcher object.

3.5. Other Compilation Issues

This section discusses two interesting compilation issues discovered in the course of implementing our MultiJava compiler.

Private external methods were mentioned in CLCM2000 but no specific compilation strategy was given. A sample of a compilation unit with a private external method is given in Figure 26. Our technique is to make the anchor class of the private external method be a nested class of the regular anchor class. For the code in the figure a `swap$anchor` nested class is created inside the `sort$anchor` class. This enforces the privileged access semantics for the private external method and avoids a name clash should a non-private external generic function named `swap` be declared in the same package.

Another issue, not addressed in CLCM2000, is the reading of MultiJava-specific features from bytecode when the MultiJava source code is not available. This is necessary, for example, when compiling a client of an external generic function when the source code is not available. It seems possible to read the bytecode for an external generic function or multimethod and then, reversing the compilation strategy, arrive at the signatures of the original MultiJava source code. However, in practice this has two distinct disadvantages. First, it is possible, though perhaps not likely, that some other tool might generate names like *ident*`$anchor` that would confuse a reverse-compilation utility. Second, because of the structure of Java bytecode [Lindholm and Yellin 2000] (§4), reading the signature of a class from its bytecode is a very efficient process. Using a reverse-compilation utility would necessar-

ily make the reading of MultiJava-specific information much more costly than reading regular Java information.

Our solution is to use the capability of adding custom attributes to bytecode [Lindholm and Yellin 2000] (§4.7.1). Using the attributes we can encode the signatures of all the local methods of an external generic function in the generic function's anchor class bytecode and encode the signatures of all internal multimethods in their receiver class bytecode. These attribute values are easily read from bytecode, allowing a MultiJava compiler to efficiently retrieve this information. Furthermore, these attributes follow a naming convention like that for Java packages [Gosling et al. 2000] (§6.8.1) in which the names are derived from an organization's internet domain name. Thus the names of our MultiJava attributes would begin "`org.multijava.`". This eliminates the possibility of name clashes with properly designed tools. Using these attributes should not cause any incompatibilities with conforming JVM implementations; such implementations must ignore attributes that are not recognized.

In CLCM2000 we mentioned that one strategy for improving the efficiency of external generic function dispatch might be to use reflection to replace the chain of dispatcher objects with a dynamically compiled global dispatching method, generated "on-the-fly". We noted that the load-time cost of this strategy might be high, but run-time invocation costs could be greatly reduced.

Having multimethod information encoded in the attributes of the bytecode files presents an intriguing new possibility. A modified JVM could read the external generic function and multimethod information from bytecode attributes and perform dispatch using native code, bypassing both the chain-of-responsibility and the cascaded instanceof tests. At the same time, retaining the chain-of-responsibility and instanceof tests within the regular bytecode would ensure that the program had the same semantics if run on a standard JVM. A similar technique of performing multiple dispatch in the JVM, though not using custom attributes, is introduced by Dutchyn, et al. [Dutchyn et al. 2001]. This technique is discussed in Section 7.

SECTION 4. IMPLEMENTATION OF MJC

This section summarizes the implementation of mjc, our MultiJava compiler. mjc is derived from the open-source Kopi Java Compiler, made available by Decision Management Systems.[27] mjc is licensed under the GNU General Public License, version 2, and can be freely downloaded.[28] The compiler consists of over 115,000 lines of Java source code, including comments. This section summarizes the implementation of mjc by sketching an outline of the architecture based on the major compilation passes, largely inherited from Kopi.[29] In reviewing the architecture we indicate the points in the control flow where additions or changes were made for MultiJava. After reviewing the basic architecture the section concludes with a discussion of the interesting changes and additions to the compiler's data structures necessary to implement MultiJava's open classes and multiple dispatch.

4.1. Compiler Architecture

The compiler architecture can be understood by understanding the basic sets of objects that are used during compilation and then by understanding how the passes of the compilation process manipulate these objects.

4.1.1. The Pieces

The classes whose instances are used during compilation can be divided into four categories. The first category of classes are those used to represent an *AST*, or abstract syntax tree. Roughly speaking, there is one AST class for each non-terminal in the combined Java and MultiJava grammar. As is typical for an AST representation, each AST class instance, or node, contains fields for recording the nodes below it in the AST. For example, a node representing a compilation unit would have fields containing nodes representing a package declaration, import statements, type declarations and augmenting method declarations. In addition to the fields used to build the structure of the AST, the AST classes also declare methods for performing operations on the trees. For example, each AST node includes a `typecheck` method. The typechecking of an AST is accomplished by invoking the `typecheck` method on the root of the AST. Each AST node is responsible for invoking the `typecheck` methods of its child nodes.

---

27. Kopi is available from `http://www.dms.at/kopi`.
28. The GNU General Public License is available from `http://www.gnu.org/copyleft/gpl.html`. mjc is currently available at `http://www.multijava.org`.
29. We have not been able to locate documentation for the Kopi compiler architecture as a whole. Thus a significant part of the effort in implementing mjc was spent in understanding the base compiler.

The second category of classes are those used to build what we call the *signature forest*. The signature forest is a set of trees, one for each type and external generic function being compiled, and one for each class or external generic function read from bytecode during compilation. Each tree represents the signature of a single class or external generic function. A global hash table maps class or generic function names to the appropriate tree in the signature forest. Thus the signature forest can be used during typechecking to find the type of a field reference or to identify the target generic function for a method invocation. In this sense the signature forest acts as a global symbol table for the compiler. In addition to this symbol-table function, the objects in the signature forest include the operations that generate bytecode at the end of the compilation process.

The third category of classes used by the compiler are the context classes. The context classes represent lexical scope and are used for control flow analysis. Each context object contains references to all the types and variables declared in that lexical context and a reference to the surrounding lexical context. Thus the context objects can act as a local symbol table. For example, when typechecking a simple name the compiler can query the local context object for the type of that name. If the name is not declared in the local context, then the local context object passes the query along to the surrounding context. Eventually this process will either identify the type of that name or reach the context object representing the entire compilation unit. This outermost context has information on the `import` statements and can attempt to resolve the simple name into a fully qualified name by consulting the signature forest.

The context classes are also used for control flow analysis. This is necessary in Java for checking things like definite assignment [Gosling et al. 2000] (§16). For example, suppose we have the following code:

```
...
int x;                      // line 1
if (checkSomething()) {     // line 2
    x = 0;
} else {
    doSomething();
}
System.out.println( x );    // line 7
```

The context representing this entire code fragment would be mutated when typechecking line 1 to record the declaration of the variable `x` and the fact that `x` is uninitialized. This context would then be used to typecheck the predicate in line 2. Assuming this check passes, then two new context objects would be created, one for checking the `if`-block and one for checking the `else`-block. The context for the `if`-block would be mutated to record that `x` is initialized. But the context for the `else`-block would

still maintain the `x` is uninitialized. After typechecking both branches, the two inner contexts would be merged with the original context. Since `x` is only initialized in one branch, the original context would now record that `x` is not definitely assigned. Thus an error would be signalled when typechecking the reference to `x` in line 7.

The fourth category of classes used by the compiler are miscellaneous utility classes. Included in this category are the classes for driving the actual compilation process, those for lexing and parsing, and those for processing error messages.

4.1.2. Assembling the Pieces

The compiler uses seven separate passes, though several of these passes only walk a portion of the AST. The first pass parses the source code and generates an AST. The second pass processes `import` statements and mutates the AST, moving internal methods declared via the augmenting method syntax into the local classes that they augment. The next three passes perform various typechecking operations. The sixth pass groups multimethods into those that will share a common dispatcher method. A final pass generates the Java bytecode.

- Parsing is performed using lexer and parser classes that are generating using ANTLR, ANother Tool for Language Recognition, published by jGuru.[30] The parser is a predicated LL(k) parser [Parr and Quong 1994].
- *Internalizing* of augmenting methods is the process of identifying augmenting methods that augment local classes and mutating the AST so that these internal methods appear within their receiver classes. To perform this internalizing operation the `import` statements must be processed so the type identifiers can be resolved to their fully qualified names. Classes and generic functions named in single-type import statements are added to the signature forest at this stage. Classes and generic functions belonging to packages imported with an import-on-demand statement are added to the signature forest lazily.
- Typechecking is performed in three passes. These can be understood in terms of the JVM execution sequence [Gosling et al. 2000] (§12). The interface checking pass is analogous to loading and linking (§12.2, §12.3). The initializers checking pass is analogous to initialization of classes and interfaces (§12.4). Finally the pass for checking instance members is analogous to creation of instances and execution (§12.5). In the typechecking passes we treat external meth-

30. ANTLR is available from `http://www.antlr.org`.

ods as belonging to the classes they augment and handle multiple dispatch as if it was part of the target language. In other words, the typechecking pass mimics the semantics of MultiJava, not the implementation.

  - Interface checking has the purpose of gathering information about type signatures so that subsequent passes can do things like finding the generic function invoked by a method call. This pass adds one tree to the signature forest for each type declaration and external generic function being compiled. Simple checks, like verifying that external methods are not abstract, are performed in this pass. This pass also adds a default constructor, if necessary, to the AST for each class and concatenates all field and instance initializers into a single initializer method.
  - Initializer checking typechecks the bodies of static initializers and records information about the initialization of static fields.
  - Instance member typechecking, appropriately enough, checks the code for all instance members. At the end of this pass the implementation-side checks for generic functions are performed.

- Multimethod grouping is performed to collect the multimethods that will share a common dispatcher method in the generated bytecode. This pass mutates the signature forest to reflect this grouping by dispatcher method. Objects are also added to the signature forest to represent the signature interfaces and nested dispatcher objects for external generic functions. After this pass the AST can be discarded.

- Code generation is accomplished by a simple walk of the trees in the signature forest.

4.2. Interesting Modifications to Support MultiJava

The previous subsection described the main data structures used by the compiler and how these data structures are manipulated by the compilation passes. This subsection highlights the modifications and additions to these data structures in `mjc` to support open classes and multimethods.

4.2.1. Handling Open Classes

The key changes to Kopi to support open classes in `mjc` are support for apparent signatures, resolution of implicit and explicit "`this`" within external methods, implicit import of external generic functions, and code generation for anchor classes, signature interfaces, and dispatcher objects.

To add support for typechecking open classes to the Kopi compiler architecture we modify the classes of the signature forest to support the notion of apparent signature. In mjc, each object representing a class in the signature forest maintains a mapping from contexts to apparent signatures. When typechecking a method reference in MultiJava the context of that reference is passed to the signature forest. This allows the reference to be resolved using the appropriate apparent signature, based on the external generic functions in scope at the reference location.

During typechecking, each object representing an external method in the signature forest is a child node of an object representing the external generic function. This external generic function object is used to generate the anchor class in bytecode.

Another modification to Kopi to support open classes is necessary to allow references to implicit or explicit "`this`" within external method bodies to resolve to the correct class. This is accomplished with a new context class that redirects name resolution from within an external method body away from the anchor class and into the receiver class in the signature forest. However, this context does not redirect all references into the external method's receiver class. For example, inner classes declared by type declaration statements or anonymous inner class declarations are added to the anchor class, since these cannot be generated in the separately-compiled receiver class.

Another challenge in handling open classes is the need to implicitly import external generic functions.[31] As with Java classes, implicit import is needed in two cases. External methods of the same package as the client are imported when referenced. External methods may also be implicitly imported from another package. This happens when two things are true: there is a package import statement in the client compilation unit and the client code references an external method defined in the imported package.

The key to implicit import is that a search for possible external methods must be performed whenever a method identifier is processed. There are two ways that a method identifier can be used, in a method declaration and in a method call. For a method declaration, the new method may be specializing an implicitly imported external method. For a method call, the method may be external. This last point is true even if an internal method of the same name and applicable static argument types exists.[32]

The final challenge in implementing open classes is the generation of the appropriate bytecode for anchor classes, signature interfaces, and dispatcher objects. As mentioned above, this is accomplished by appropriate mutations of the signature forest following typechecking. Several new signature forest

31. As of this writing, the implementation of these features in mjc is not finished.
32. In this case Java's usual rules for selecting between applicable generic functions apply [Gosling et al. 2000] (§15.12.2).

subclasses specialize the behavior of Kopi signature forest classes to implement the new code generation.

4.2.2. Handling Multimethods

The key new features in mjc to support multimethods are specialized parameter types, new signature forest classes to generate dispatcher methods, and enhancements to the typechecking code to support the restrictions of Subsection 2.4.

In mjc, the classes representing parameters in the AST include information on both the static parameter type and the explicit specializer. The corresponding classes in the signature forest contain both static and dynamic type information. The static information is used for generic function selection and multimethod grouping. The dynamic information is used for ambiguity checks and generation of dispatcher methods and multimethod bodies in bytecode.

Most of the typechecking requirements of Subsection 2.4 are implemented in mjc via simple additions to the original Kopi typechecking code. The exception is the implementation side typechecking of entire generic functions. These checks are performed by a substantial body of code added to the typechecking of individual type declarations. This code is a straightforward implementation of the algorithm given in “Checks on Entire Generic Functions” on page 38, extended to consider the restrictions given in Subsection 2.4.2.

SECTION 5. EVALUATION

This section evaluates the MultiJava language design and the performance of code generated by `mjc`. The discussion, like the evaluation, is divided into four part. The first part examines a series of interpreters, written in MultiJava, for the untyped lambda calculus. The second part compares the performance of an algorithm implemented using the extensible visitor pattern against the same algorithm implemented using MultiJava's open classes. The third part examines the performance of an algorithm implemented using regular Java typecases and MultiJava's multiple dispatch. A fourth part compares the cost of the modular solutions versus non-modular solutions to quantify the cost of modularity. We conclude by revisiting the design constraints introduced in Subsection 1.2.

5.1. Writing and Extending Interpreters Using MultiJava

To evaluate the expressiveness and flexibility of MultiJava we implemented a series of interpreters for the untyped lambda calculus [Barendregt 1984, Schmidt 1994].[33] The concrete syntax that we use is taken from Scheme [Kelsey et al. 1998]. The first of these interpreters, with source code[34] shown in Figures 27 through 32, is for the language:

Term:
    Variable
    Lambda
    Application
Variable:
    String
Lambda:
    `(` `lambda` `(` *String*$_{opt}$ `)` *Term* `)`
Application:
    `(` *Term* *Terms*$_{opt}$ `)`
Terms:
    Term
    Term *Terms*
String:
    any legal Java string literal

The `Interpreter1` class shown in Figure 27 takes a *Term*, constructs a new empty environment, and reduces the term[35]. The *Term*, *Variable*, *Lambda*, and *Application* non-terminals are represented by

33. This example was motivated by Essentials of Programming Languages [Friedman et al. 1992]. Zenger and Odersky use the same example [Zenger and Odersky 2001]; this facilitates comparison of their work with ours.
34. Source code shown in this section generally omits instance fields and accessor methods where they can be inferred from the given constructors.
35. For the interested, the term is reduced to weak head-normal form using a call-by-value semantics.

```
// compilation unit "Interpreter1.java"
package evaluation;

// Interpreter for the simple untyped lambda calculus.
public class Interpreter1 {
    public void interpret( Term ast ) {
        Environment env = initialEnvironment();
        try {
            Term result = ast.eval( env );
            System.out.println( result.toString() );
        } catch (EvaluationException e) {
            System.out.println( "error:  " + e.getMessage() );
        } // end of try-catch
    }

    protected Environment initialEnvironment() {
        return new Environment();
    }

}
```

Figure 27: Interpreter for the untyped lambda calculus

```
// compilation unit "Term.java"
package evaluation;

public abstract class Term {
    public Term eval( Environment e ) throws EvaluationException {
        return this;
    }
}
```

Figure 28: Abstract class representing the type of all terms in the lambda calculus

```
// compilation unit "Lambda.java"
package evaluation;

public class Lambda extends Term {

    public Lambda( String[] formals, Term body ) {
        this.formals = formals;
        this.body = body;
    }

    // inherit eval

    /* ... */
}
```

Figure 29: Class representing lambda expressions in the lambda calculus

corresponding classes. Each of these classes includes an `eval` method that takes an `Environment` argument. The default behavior, specified in the `Term` abstract class of Figure 28, is to return the term

```
// compilation unit "Variable.java"
package evaluation;

public class Variable extends Term {
    public Variable( String name ) {
        this.name = name;
    }

    public Term eval( Environment e ) throws EvaluationException {
        Term result = e.valueOf( name );
        return result == null ? this : e.valueOf( name );
    }
    /* ... */
}
```

Figure 30: Class representing variables in the lambda calculus

```
// compilation unit "Application.java"
package evaluation;

public class Application extends Term {
    public Application( Term rator, Term[] rands ) {
        this.rator = rator;
        this.rands = rands;
    }

    public Term eval( Environment e ) throws EvaluationException {
        try {
            Lambda function = (Lambda) rator.eval( e );
            String[] formals = function.formals();
            if (formals.length != rands.length) {
                throw new EvaluationException( "Number of formals (" +
                                               function.formals().length +
                                               ") different than number " +
                                               "of actuals (" + rands.length +
                                               ")" );
            }
            Term[] actuals = new Term[ rands.length ];
            for (int i = 0; i < rands.length; i++) {
                actuals[i] = rands[i].eval( e );
            } // end of for
            Environment evalEnv = e;
            for (int i = 0; i < rands.length; i++) {
                evalEnv = evalEnv.bind( formals[i], actuals[i] );
            }
            return function.body().eval( evalEnv );
        } catch (ClassCastException ce) {
            throw new EvaluationException( "Operator is not a lambda " +
                                           "expression" );
        }
    }
    /* ... */
}
```

Figure 31: Class representing applications in the lambda calculus

```
// compilation unit "Environment.java"
package evaluation;
import java.util.HashMap;

public class Environment {

    public Environment() {
        this( new HashMap() );
    }

    protected Environment( HashMap map ) {
        this.map = map;
    }

    // factory method
    protected Environment makeInstance( HashMap map ) {
        return new Environment( map );
    }

    public Environment bind( String name, Term value ) {
        HashMap newMap = (HashMap) map.clone();
        newMap.put( name, value );
        return makeInstance( newMap );
    }

    public Term valueOf( String name ) throws EvaluationException {
        return (Term) map.get( name );
    }

    protected HashMap map;
}
```

Figure 32: Class representing environments mapping for names to terms

unchanged. This behavior is inherited by the `Lambda` class in Figure 29. For `Variable`s, declared in Figure 30, the `eval` method simply looks up the value of the variable in the given environment. If the variable is undefined in the given environment then the variable itself is returned unreduced. The eval method of the `Application` class, declared in Figure 31, reduces the operator and each operand in the original environment. Next the formal parameters of the operand term are bound to the values of the operands. Finally the body of the operator is reduced in the new environment. The Environment class is declared in Figure 32. The `Driver1` class, shown in Figure 33, constructs several terms in the language and uses the `Interpreter1` class to evaluate them, with the following results:

```
$ java evaluation.Driver1
evaluation.Variable@4b222f
evaluation.Variable@3169f8
evaluation.Lambda@2457b6
evaluation.Variable@7a78d3
```

```
// compilation unit "Driver1.java"
package evaluation;

public class Driver1 {
    public static void main (String[] args) {
        test( new Interpreter1() );
    }

    public static void test( Interpreter1 i ) {
        i.interpret( VARREF_X );
        i.interpret( VARREF_Y );
        i.interpret( ID );
        i.interpret( APP1 );
        i.interpret( APP2 );
        i.interpret( APPFUNC );
        i.interpret( APP3 );
        i.interpret( APP4 );
    }

    // x
    public static Term VARREF_X = new Variable( "x" );

    // y
    public static Term VARREF_Y = new Variable( "y" );

    // (lambda (y) y)
    public static Term ID =
        new Lambda( new String[] { "y" }, VARREF_Y );

    // ((lambda (y) y) x)
    public static Term APP1 =
        new Application( ID, new Term[] { VARREF_X } );

    // ((lambda (y) y) (lambda (y) y))
    public static Term APP2 =
        new Application( ID, new Term[] { ID } );

    // (lambda (f x) (f x))
    public static Term APPFUNC =
        new Lambda( new String[] { "f", "x" },
                    new Application( new Variable( "f" ),
                                     new Term[] { VARREF_X } ) );

    // ((lambda (f x) (f x)) (lambda (y) y) x)
    public static Term APP3 =
        new Application( APPFUNC, new Term[] { ID, VARREF_X } );

    // ((lambda (f x) (f x)) (lambda (y) y) (lambda (y) y))
    public static Term APP4 =
        new Application( APPFUNC, new Term[] { ID, ID } );

}
```

Figure 33: Test cases for the interpreter of Figure 27

```
// compilation unit "prettyPrint.java"
package evaluation;

include evaluation.prettyPrint;

public String Term.prettyPrint() {
    return "<undefined>";
}

public String Variable.prettyPrint() {
    return name();
}

public String Lambda.prettyPrint() {
    StringBuffer result = new StringBuffer( "(lambda (" );
    String[] formals = formals();
    for (int i = 0; i < formals.length; i++) {
        result.append( formals[i] );
        if (i < formals.length - 1 ) {
            result.append( " " );
        } // end of if
    } // end of for

    result.append( ") " );
    result.append( body().prettyPrint() );
    result.append( ")" );
    return result.toString();
}

public String Application.prettyPrint() {
    StringBuffer result = new StringBuffer( "(" );
    result.append( rator().prettyPrint() + " " );
    Term[] rands = rands();
    for (int i = 0; i < rands.length; i++) {
        result.append( rands[i].prettyPrint() );
        if (i < rands.length - 1) {
            result.append( " " );
        } // end of if
    } // end of for
    result.append( ")" );
    return result.toString();
}
```

Figure 34: External `prettyPrint` generic function for lambda calculus terms

```
evaluation.Lambda@2457b6
evaluation.Lambda@129206
evaluation.Variable@30f13d
evaluation.Lambda@2457b6
```

Because `Term` and its subclasses did not override the `Object.toString` method, the results of evaluating the various terms are given as class names and object hash codes. The next interpreter rectifies this by adding an external `prettyPrint` generic function, declared in Figure 34. An extended

```
// compilation unit "Interpreter2.java"
package evaluation;

include evaluation.prettyPrint;

public class Interpreter2 extends Interpreter1 {
    public void interpret( Term ast ) {
        Environment env = initialEnvironment();
        System.out.print( ast.prettyPrint() + " ==> " );
        try {
            Term result = ast.eval( env );
            System.out.println( result.prettyPrint() );
        } catch (EvaluationException e) {
            System.out.println( "error:  " + e.getMessage() );
        }

    }
}
```

Figure 35: Interpreter using `prettyPrint` external generic function

```
// compilation unit "Driver2.java"
package evaluation;

public class Driver2 extends Driver1 {
    public static void main (String[] args) {
        test( new Interpreter2() );
    }
}
```

Figure 36: Test driver for the interpreter of Figure 35

interpreter using the new generic function is given in Figure 35 and a driver for running the test cases with the new interpreter is given in Figure 36. The results for this interpreter are:

```
$ java evaluation.Driver2
x ==> x
y ==> y
(lambda (y) y) ==> (lambda (y) y)
((lambda (y) y) x) ==> x
((lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
(lambda (f x) (f x)) ==> (lambda (f x) (f x))
((lambda (f x) (f x)) (lambda (y) y) x) ==> x
((lambda (f x) (f x)) (lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
```

The next interpreter extends the interpreted language to add numbers and addition. The changes to the language are:

Term:
...
Number
Plus

```
// compilation unit "Number.java"
package evaluation;

include evaluation.prettyPrint;

public class Number extends Term {

    public Number( int val ) {
        this.val = val;
    }

    // inherits eval method

    public String prettyPrint() {
        return "" + val;
    }

    /* ... */
}
```

Figure 37: Class representing numbers in the extended lambda calculus

Number:
any legal Java integer literal
Plus:
`(` `+` $Terms_{opt}$ `)`

This interpreter demonstrates overriding of external methods by new subclasses. For example, the `prettyPrint` methods in `Number` (see Figure 37) and `Plus` (see Figure 38) belong to the external generic function declared in Figure 34. `Number` inherits the default `eval` method from `Term`, while `Plus` declares its own `eval` method. Also interesting is the fact that, because of subtype polymorphism, a new interpreter class is not needed to evaluate the extended language. Of course a new test driver is needed to exercise the new variants. The code for this driver appears in Figure 39 and the results for this new language are:

```
$ java evaluation.Driver3
x ==> x
y ==> y
(lambda (y) y) ==> (lambda (y) y)
((lambda (y) y) x) ==> x
((lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
(lambda (f x) (f x)) ==> (lambda (f x) (f x))
((lambda (f x) (f x)) (lambda (y) y) x) ==> x
((lambda (f x) (f x)) (lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
0 ==> 0
1 ==> 1
(+) ==> 0
(+ 1) ==> 1
```

```
// compilation unit "Plus.java"
package evaluation;
include evaluation.prettyPrint;

public class Plus extends Term {
    public Plus( Term[] rands ) {
        this.rands = rands;
    }

    public Term eval( Environment env ) throws EvaluationException {
        int result = 0;
        for (int i = 0; i < rands.length; i++) {
            Term value = rands[i].eval( env );
            try {
                result += ((Number) value).val();
            } catch (ClassCastException e) {
                throw new EvaluationException( "Non-number operand \"" +
                                               rands[i].prettyPrint() +
                                               "\" as argument of plus " +
                                               "operator" );
            }
        }
        return new Number( result );
    }

    public String prettyPrint() {
        StringBuffer result = new StringBuffer( "(+" );
        for (int i = 0; i < rands.length; i++) {
            result.append( " " + rands[i].prettyPrint() );
        }
        result.append( ")" );
        return result.toString();
    }

    /* ... */
}
```

Figure 38: Class representing addition operator in the extended lambda calculus

```
(+ 1 1) ==> 2
(lambda (x) (+ x 1)) ==> (lambda (x) (+ x 1))
((lambda (x) (+ x 1)) 1) ==> 2
((lambda (x) (+ x 1)) (+ 1 1)) ==> 3
```

The final interpreter example introduces sequences, and assignment to the language:

Term:
...
Sequence
Assignment

Sequence:
(begin *Term* *Terms*$_{opt}$)

```
// compilation unit "Driver3.java"
package evaluation;

public class Driver3 extends Driver1 {
    public static void main (String[] args) {
        Interpreter2 i = new Interpreter2();
        test( i );
        testNumber( i );
    }

    public static void testNumber( Interpreter2 i ) {
        i.interpret( ZERO );
        i.interpret( ONE );
        i.interpret( ZERO_B );
        i.interpret( ONE_B );
        i.interpret( TWO );
        i.interpret( ADD_ONE );
        i.interpret( TWO_B );
        i.interpret( THREE );
    }

    // 0
    public static Term ZERO = new Number( 0 );

    // 1
    public static Term ONE = new Number( 1 );

    // (+)
    public static Term ZERO_B = new Plus( new Term[0] );

    // (+ 1)
    public static Term ONE_B = new Plus( new Term[] { ONE } );

    // (+ 1 1)
    public static Term TWO = new Plus( new Term[] { ONE, ONE } );

    // (lambda (x) (+ x 1))
    public static Term ADD_ONE =
        new Lambda( new String[] { "x" },
                    new Plus( new Term[] { VARREF_X, ONE } ) );

    // ((lambda (x) (+ x 1)) 1)
    public static Term TWO_B =
        new Application( ADD_ONE, new Term[] { ONE } );

    public static Term THREE =
        new Application( ADD_ONE, new Term[] { TWO } );
}
```

Figure 39: Test cases for the lambda calculus extended with numbers and addition

```
// compilation unit "EnvironmentWithStore.java"
package evaluation;
import java.util.HashMap;

public class EnvironmentWithStore extends Environment {
    public EnvironmentWithStore() {
        super();
    }

    protected EnvironmentWithStore( HashMap map ) {
        super(map);
    }

    // factory method
    protected Environment makeInstance( HashMap map ) {
        return new EnvironmentWithStore( map );
    }

    public Term updateLocation( String name, Term value )
        throws EvaluationException
    {
        if (map.get(name) == null) {
            throw new EvaluationException( "unable to mutate undefined " +
                                           "variable \"" + name + "\"" );
        }
        map.put( name, value );
        return value;
    }
}
```

Figure 40: Class extending `Environment` to support mutation

```
// compilation unit "Interpreter4.java"
package evaluation;

include evaluation.prettyPrint;

public class Interpreter4 extends Interpreter2 {
    public Environment initialEnvironment() {
        return new EnvironmentWithStore();
    }
}
```

Figure 41: Interpreter for the lambda calculus extended with sequences and assignment

Assignment:
`( set!` *String Term* `)`

An `EnvironmentWithStore` class is introduced in Figure 40. A new interpreter that uses environments with stores is given in Figure 41. Only the `initialEnvironment` method of `Interpreter2` needs to be overridden to arrive at `Interpreter4`. The `Sequence` class is given in Figure 42. The terms in a sequence are evaluated sequentially and the value of the sequence is the value of the last

```
// compilation unit "Sequence.java"
package evaluation;
include evaluation.prettyPrint;

public class Sequence extends Term {

    public Sequence( Term[] terms ) {
        this.terms = terms;
    }

    public String prettyPrint() {
        StringBuffer result = new StringBuffer( "(begin" );
        for (int i = 0; i < terms.length; i++) {
            result.append( " " + terms[i].prettyPrint() );
        } // end of for
        result.append( ")" );
        return result.toString();
    }

    public Term eval( Environment env ) throws EvaluationException {
        if (terms.length == 0) {
            throw new EvaluationException( "empty sequences not allowed" );
        } // end of if
        Term result = terms[0].eval( env );
        for (int i = 1; i < terms.length; i++) {
            result = terms[i].eval( env );
        } // end of for
        return result;
    }

    /* ... */
}
```

Figure 42: Class representing sequences in the extended lambda calculus

term in the sequence. An interesting aspect of this interpreter is the use of multiple dispatch in the `eval` method of the `Assignment` class, given in Figure 43. If an environment with a store is passed to the `eval` method then the cell referenced by the given name is mutated to contain the new value. On the other hand, if an environment without a store is passed to the `eval` method, then the default behavior, inherited from `Term`, is used; i.e., the assignment expression is returned unchanged since it cannot be reduced in the given environment. The class `Driver4`, given in Figure 44, uses both `Interpreter2` and `Interpreter4` for evaluation to demonstrate this distinction. We have the following results.

```
$ java evaluation.Driver4
x ==> x
y ==> y
(lambda (y) y) ==> (lambda (y) y)
((lambda (y) y) x) ==> x
((lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
```

```
// compilation unit "Assignment.java"
package evaluation;

include evaluation.prettyPrint;

public class Assignment extends Term {
    public Assignment( String name, Term expr ) {
        this.name = name;
        this.expr = expr;
    }

    public String prettyPrint() {
        return "(set! " + name + " " + expr.prettyPrint() + ")";
    }

    // inherit the default (i.e. do-nothing) eval method since we
    // can only have side-effects if we have a store
    public Term eval( Environment@EnvironmentWithStore env )
        throws EvaluationException
    {
        return env.updateLocation( name, super.eval( env ) );
    }

    /* ... */
}
```

Figure 43: Class representing assignment in the extended lambda calculus

```
(lambda (f x) (f x)) ==> (lambda (f x) (f x))
((lambda (f x) (f x)) (lambda (y) y) x) ==> x
((lambda (f x) (f x)) (lambda (y) y) (lambda (y) y)) ==> (lambda (y) y)
0 ==> 0
1 ==> 1
(+) ==> 0
(+ 1) ==> 1
(+ 1 1) ==> 2
(lambda (x) (+ x 1)) ==> (lambda (x) (+ x 1))
((lambda (x) (+ x 1)) 1) ==> 2
((lambda (x) (+ x 1)) (+ 1 1)) ==> 3
using environment with store:
--------------------------
(begin) ==> error:  empty sequences not allowed
(begin x) ==> x
(begin x y) ==> y
(set! x 1) ==> error:  unable to mutate undefined variable "x"
(lambda (x y) (+ x y)) ==> (lambda (x y) (+ x y))
(lambda (x y) (begin (set! x y) (+ x y))) ==> (lambda (x y) (begin (set! x y)
                                                                  (+ x y)))
((lambda (x y) (+ x y)) 1 (+ 1 1)) ==> 3
((lambda (x y) (begin (set! x y) (+ x y))) 1 (+ 1 1)) ==> 4
using environment without store:
---------------------------
```

```
// compilation unit "Driver4.java"
package evaluation;
public class Driver4 extends Driver3 {
    public static void main (String[] args) {
        Interpreter4 i = new Interpreter4();
        test( i );testNumber( i );
        System.out.println();
        System.out.println("using environment with store:");
        System.out.println("--------------------------");
        testSequence( i );
        System.out.println();
        System.out.println("using environment without store:");
        System.out.println("-----------------------------");
        testSequence( new Interpreter2() );
    }
    public static void testSequence( Interpreter2 i ) {
        i.interpret( VARREF_SEQ );
        i.interpret( VARREFS_SEQ );
        i.interpret( BAD_ASSN );
        i.interpret( SUM );
        i.interpret( TIMES_TWO );
        i.interpret( APP_SUM );
        i.interpret( APP_TIMES_TWO );
    }
    // (begin x)
    public static Term VARREF_SEQ = new Sequence( new Term[] { VARREF_X } );

    // (begin x y)
    public static Term VARREFS_SEQ = new Sequence( new Term[] { VARREF_X,
                                                                VARREF_Y } );
    // (set x 1)
    public static Term BAD_ASSN = new Assignment( "x", ONE );

    // (lambda (x y) (+ x y))
    public static Term SUM =
        new Lambda( new String[] { "x", "y" },
                    new Plus( new Term[] { VARREF_X, VARREF_Y } ) );

    // (lambda (x y) (begin (set x y) (+ x y)))
    public static Term TIMES_TWO =
        new Lambda( new String[] { "x", "y" },
                    new Sequence( new Term[] {
                                    new Assignment( "x", VARREF_Y ),
                                    new Plus( new Term[] { VARREF_X,
                                                           VARREF_Y } ) } ));

    // ((lambda (x y) (+ x y)) 1 2)
    public static Term APP_SUM = new Application(SUM, new Term[] {ONE, TWO});

    // ((lambda (x y) (begin (set x y) (+ x y))) 1 2)
    public static Term APP_TIMES_TWO =
        new Application( TIMES_TWO, new Term[] { ONE, TWO } );
}
```

Figure 44: Test cases for the lambda calculus extended with sequences and assignment

```
(begin) ==> error:  empty sequences not allowed
(begin x) ==> x
(begin x y) ==> y
(set! x 1) ==> (set! x 1)
(lambda (x y) (+ x y)) ==> (lambda (x y) (+ x y))
(lambda (x y) (begin (set! x y) (+ x y))) ==> (lambda (x y) (begin (set! x y)
                                                                  (+ x y)))
((lambda (x y) (+ x y)) 1 (+ 1 1)) ==> 3
((lambda (x y) (begin (set! x y) (+ x y))) 1 (+ 1 1)) ==> 3
```

5.2. Open Class Performance

To evaluate the performance of code compiled by mjc that uses the open class mechanism, we implemented a simple binary tree in Java and extended this tree by adding three different external generic functions and then a new subclass representing n-ary trees. We also implemented the same code and extensions using the extensible visitor pattern [Krishnamurthi et al. 1998]. As the code for these tests is quite similar to the examples discussed previously, it has been mostly relegated to Appendix A.1. In both implementation styles the first operation is a simple tree walk that performs no additional calculations. By timing this operation we can measure the relative dispatch times of the two implementation styles. The second operation calculates the size of the tree. Figure 45 gives the code for this operation in the open class implementation; Figure 46 does the same for the extensible visitor implementation. The third operation returns a string representation of the tree. By timing these later two operations we can measure the performance of calculations that must be written differently to accommodate the two implementation styles. (The primary implementation differences being the need to pass arguments and return results through the state of the visitor, as discussed on page 5, and the need for factory methods to allow the visitors to be extended when subclasses are added.) Appendix A.3 includes the raw data from our testing.

Table 2 on page 86 gives the results for the simple tree walk operation.[36] For each implementation style, the operation is invoked 100,000 times on each of three different trees. The first tree is a simple binary tree with just 5 nodes, the second is an n-ary tree with 7 nodes. The final tree is an n-ary tree of depth 4 and branching factor 4, containing 341 nodes. These results clearly demonstrate that pure dispatch speed for the open class implementation is dramatically slower than for the extensible visitor code.

But the information on dispatch speed does not tell the complete story. The extensible visitor pattern requires substantial additional code to implement the actual calculations within the methods. As

36. All test results are from the Sun JDK 1.3.1 using the HotSpot JVM under Windows 2000 Professional on a Dell Inspiron 5000e with an 850 MHz Intel Pentium III processor and 256 MB of physical RAM.

```
// compilation unit "size.java"
package evaluation.speed;

public int Tree.size() {
    return 1;
}

public int Interior.size() {
    return 1 + left().size() + right().size();
}
```

```
// compilation unit "MultiInterior.java"
package evaluation.speed;

public class MultiInterior extends Tree {
    public MultiInterior( Object value, Tree[] children ) {
        super( value );
        this.children = children;
    }

/* code for dispatchTest and prettyPrint omitted */

    public int size() {
        int result = 1;
        for (int i = 0; i < children.length; i++) {
            result += children[i].size();
        }
        return result;
    }

    public Tree[] children() {
        return children;
    }

    private Tree[] children;
}
```

Figure 45: Open class implementation of a tree size operation

shown in Figure 46, extensible visitor requires factory methods to generate the appropriate visitor for recursive calls, often with the attendant copying of state to or from the new visitor. Table 3 on page 86 shows the results for calculating tree size, using the same trees as above. We see that once calculations are considered the speed advantage of extensible visitor is approximately halved. This is due to the additional complexity of the extensible visitor pattern. In fact, if we compare results from Tables 2 and 3 we see that introducing calculations in the extensible visitor implementation nearly doubles its execution time, while the calculations in the open class code result in essentially no change versus the simple tree walk. We should note, however, that it would certainly be possible to improve the efficiency of the extensible visitor code in this example by mutating the state of the visitor instead of creating new

```
// compilation unit "VTreeSizer.java"
package evaluation.speed;

public class VTreeSizer implements VTreeVisitor {
    public VTreeSizer() {
        size = 0;
    }

    public VTreeVisitor makeInstance() {   // factory method
        return new VTreeSizer();
    }

    public void visitVTree( VTree tree ) {
        size = 1;
    }

    public void visitVInterior( VInterior interior ) {
        VTreeSizer lSizer = (VTreeSizer) makeInstance();
        VTreeSizer rSizer = (VTreeSizer) makeInstance();

        interior.left().accept( lSizer );
        interior.right().accept( rSizer );
        size = 1 + lSizer.result() + rSizer.result(); // copying state
    }

    public int result() {
        return size;
    }
    protected int size;
}
```

```
// compilation unit "VMultiTreeSizer.java"
package evaluation.speed;

public class VMultiTreeSizer extends VTreeSizer implements VMultiTreeVisitor {
    public VMultiTreeSizer() {
        super();
    }

    public VTreeVisitor makeInstance() {   // factory method
        return new VMultiTreeSizer();
    }

    public void visitVMultiInterior( VMultiInterior interior ) {
        VTree[] children = interior.children();
        size = 1;
        for (int i = 0; i < children.length; i++) {
            VTreeSizer sizer = (VTreeSizer) makeInstance();
            children[i].accept(sizer);
            size += sizer.result();
        } // end of for
    }
}
```

Figure 46: Extensible visitor implementation of a tree size operation

Table 2: Comparison of dispatch times for simple tree walk

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	50 ms[b]	80 ms	3,265 ms
Open Classes	270 ms	311 ms	15,542 ms
Speed up[a]	0.19	0.26	0.21

a. extensible visitor time / open classes time
b. times in milliseconds for 100,000 invocations

Table 3: Comparison of dispatch times for tree size calculation

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	120 ms	160 ms	7,461 ms
Open Classes	251 ms	310 ms	15,783 ms
Speed up	0.48	0.52	0.47

Table 4: Comparison of dispatch times for pretty print operation

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	1,792 ms	2,254 ms	141,804 ms
Open Classes	1,322 ms	1,763 ms	129,135 ms
Speed up	1.36	1.28	1.10

visitor instances. On the other hand, in general the extensible visitor pattern may require the creation of new visitor instances during the recursion over a data structure. Therefore these results can be considered representative of cases that do occur in practice.

The final set of tests for open class dispatch performance involve even more complex calculations. Table 4 gives the results for 100,000 invocations of a pretty-printing operation on each of our

three sample trees. In this case the additional complexity of the extensible visitor pattern swamps the dispatch speed disadvantages of open classes. For the pretty print operation the open class implementation is actually faster.

The open class implementation is much clearer and less error-prone than the extensible visitor implementation. We have demonstrated that the complexity of calculations in the extensible visitor implementation can easily overwhelm its raw dispatch speed advantages. And, as argued in Section 1.1, the visitor pattern requires advance planning and so may not even be possible in some circumstances. Finally we have not really begun to explore possibilities for optimizing the compilation strategy for open classes and a more efficient strategy may be possible (for example, using the dynamic compilation strategy mentioned on page 62).

5.3. Multiple Dispatch Performance

To evaluate the performance of code compiled by `mjc` that uses the multiple dispatch mechanism, we implemented classes representing the real numbers, the integers, and the rationals. We implemented the binary operation, multiply, on instances of these classes, using multimethods to maintain the highest possible precision in the results. For example, the product of two rational numbers is stored as a rational number. We also implemented the operation using typecases. The complete source code of the tests is given in Appendix A.2. (Double-dispatching was not used here because it requires non-modular editing to solve the binary method problem. We evaluated non-modular solutions, including double-dispatching, in the next subsection.)

To measure the dispatch speed we instantiated one real, one integer, and one rational and invoked the multiply operation 1,000,000 times on each possible combination (for a total of 9,000,000 invocations.) This test was repeated for both implementations; the results appear in Table 6. The table shows that the multiple dispatch and typecases approach yield the same performance. This is somewhat surprising. Although the generated code for multimethod dispatch in `mjc` should match the programmer coded typecase, we would expect some penalty in MultiJava for dispatch to the `mjc`-generated *ident*`$body` multimethod bodies. We suspect that the JIT compiler within the HotSpot JVM has the effect of dynamically inlining the multimethod bodies.

5.3.1. The Price of Modularity

The performance comparisons in the previous subsections pit MultiJava against regular Java implementations that offer the same degree of modularity, albeit with greater programmer effort and risk of error. MultiJava compares favorably in these experiments.

Table 5: Comparison of dispatch times for multiply operation

Implementation	Time[b]
Typecases	4,307 ms
Multiple Dispatch	4,306 ms
Speed Up[a]	1.00

a. typecases time / multiple dispatch time
b. times in milliseconds for 9,000,000 invocations

It is also interesting to investigate the cost of this modularity. We do this by comparing the results of the previous subsections against non-extensible implementations of the same algorithms. This is an "apples-to-oranges" comparison, but it provides valuable information, particularly for choosing implementation strategies when one knows that the code will not have to be extended.[37]

Table 6 compares the performance of the pretty-print operation for trees implemented in regular Java code, using the visitor pattern, using extensible visitor, and using an external generic function in MultiJava. The intent of these tests is to compare the cost of modularity for various partial solutions to the augmenting method problem. The regular Java implementation does not solve the augmenting method problem but does provide a basis for comparing the other solutions. As discussed previously, the visitor pattern is a partial solution in that it does not permit the modular addition of new classes, only new operations. Extensible visitor is more modular, allowing the modular addition of both new classes and new operations, but only if the original implementation included the necessary infrastructure (e.g., accept methods and factory methods within visitors). Open classes are the most modular, allowing the modular addition of new classes and new operations without the need for advance planning. The results show that MultiJava's open classes are more efficient than even the partial solutions of visitor and extensible visitor. This difference must be due to the additional complexity created by copying state between visitors and retrieving results via a separate method invocation.

Table 7 compares the performance of the multiply operations implemented using MultiJava multimethods and using double-dispatch. The table shows that multiple dispatch is substantially slower than double-dispatching. This is most likely because, in double-dispatching, the dispatch on each parameter

37. The author steadfastly refuses to insert a snide comment here.

Table 6: Comparison of augmenting method dispatch times for varying degrees of modularity

Implementation	Time[a]	Speed Up[b]
Regular Methods	1,061 ms	–
Visitor Pattern	1,703 ms	0.62
Extensible Visitor	1,792 ms	0.59
Open Classes	1,322 ms	0.80

a. times in milliseconds for 100,000 invocations of a prettyPrint operation on a 5 node binary tree
b. regular method time / given implementation's time

Table 7: Comparison of binary method dispatch times for varying degrees of modularity

Implementation	Time[b]
Double-dispatching	2,704 ms
Multiple Dispatch	4,306 ms
Speed Up[a]	0.63

a. double-dispatching time / multiple dispatch time
b. times in milliseconds for 9,000,000 invocations

is performed in native code within the JVM. Since mjc targets a standard JVM, dispatch on the non-receiver parameters of a multimethod is performed in Java bytecode. These test results indicate that a custom JVM for MultiJava, as discussed on page 62, might provide substantial performance benefits. Although the double-dispatching technique is faster than the current implementation of multimethods, double-dispatching is tedious, error-prone, and non-modular, as noted in Subsection 1.1.2.

5.4. Goals Revisited

Subsection 1.2 on page 13 introduced the goals of this work and a set of constraints under which MultiJava was developed. The interpreter presented in Subsection 5.1 demonstrates that MultiJava solves the augmenting method problem. The binary method examples in Subsection 5.3 demonstrate

that MultiJava solves the binary method problem. Our mjc compiler demonstrates that MultiJava satisfies its first three design constraints:

- MultiJava provides complete backward compatibility with the extant Java language.
- The modular static typechecking and compilation properties of Java are maintained.
- Output of mjc targets the standard JVM.

The fourth design constraint has to due with the efficiency of generated code. While designing MultiJava this was not a primary concern, the focus being placed on expressiveness and modularity. Nonetheless, the results given above demonstrate that MultiJava code compares favorably with code implemented using classical single dispatch techniques. As with any benchmarking, the results will vary with the data. For example, because of the chain of responsibility pattern, we can expect the dispatch cost for external methods to scale linearly with the number of internal methods added to an external generic function. With the extensible visitor pattern the dispatch cost doesn't vary with the number of methods added but with the number of visitor extensions.

SECTION 6. DESIGN ALTERNATIVES AND EXTENSIONS

This section briefly describes several potential extensions to the MultiJava language and the challenges each would pose. While a full elaboration of these extensions is beyond the scope of this work, we present a sketch of how each extension could be achieved. Following the discussion of extensions is a description of TupleJava, an earlier approach to adding open classes and multiple dispatch to Java. TupleJava is interesting in that it sacrifices some of the flexibility of MultiJava in exchange for a conceptual model that is arguably clearer.

6.1. Extending MultiJava

Most of the extensions to MultiJava addressed here involve relaxing restrictions currently placed on the language. We chose to begin with the most restricted version of the language based on Barbara Liskov's advice that the only way to know if a restriction is tolerable is by trying it.[38]

6.1.1. Overridden Method Invocations

The restriction that a method invoking `overriddenMethod()` have a single directly overridden method could be relaxed by extending the syntax. Specializers could be added to the argument expressions in an overridden method invocation to disambiguate between several directly overridden methods. For example, suppose we have the following external method declarations:

```
public Shape Shape.union(Shape s) { ... }             // method 1
public Shape Shape.union(Shape@Rectangle r) { ... }   // method 2
public Shape Rectangle.union(Shape s) { ... }         // method 3
public Shape Rectangle.union(Shape@Rectangle r ) {    // method 4
    ...
    Shape result = overriddenMethod(r);
    ...
}
```

The overridden method invocation in this example is ambiguous between methods 2 and 3 and so would be statically rejected. By adding disambiguating syntax this invocation could be rewritten:

```
Shape result = overriddenMethod(r@Rectangle);
```

This new overridden method invocation unambiguously targets method 2.

Implementing this new syntax would be a simple matter of adding the appropriate rules to the parser grammar and modifying the typechecking code for overridden method to select the appropriate target multimethod based on the specializers.

38. As recalled by Gary Leavens.

6.1.2. Self-augmenting Classes

The prohibition on self-augmenting classes could be relaxed (see “Self-augmenting Classes” on page 21). However, relaxing this restriction would add yet another wrinkle to the compilation of superclass method invocations (see Subsection 3.4.1 on page 56). In cases ***s-i3*** and ***s-e3*** of the compilation strategy we note that a superclass method invocation may target an external method that is not the first matching method in the chain of responsibility.[39] With the prohibition on self-augmenting classes the earlier matching method in the chain must be a local declaration. The `old_function` field is used to bypass the earlier matching method in these cases. However, if the prohibition is lifted a non-local declaration might exist for the earlier matching method in the chain. That is, the class containing the superclass method invocation might be self-augmenting with the target generic function. The invocation must skip the methods of the target generic function with receivers matching the self-augmenting class. The solution is indicated by noting that ***R3*** ensures the methods to be skipped are external and declared in the same compilation unit as the target generic function’s top method. Case ***s-e4*** describes a technique for handling superclass method invocations in just that context. Namely, an additional dispatcher method is introduced “that only dispatches to multimethod bodies whose receiver class is a proper supertype of the caller’s.” With self-augmenting classes these additional dispatchers would have to be introduced for all possible unseen superclass method invocations.

However, this need for additional dispatchers points out an encapsulation problem that is the dual of the one for superclass method invocations from external methods to different generic functions (see “Superclass Method Invocations” on page 32). Figure 47 shows code like the example from Figure 12 but using external methods. The code in part b) of the figure breaks the encapsulation of the generic function in part a) by explicitly invoking the superclass method invocation entry point and skipping the appropriate target method.

Thus, if the prohibition on self-augmenting classes were lifted, we would need to prohibit superclass method invocations from self-augmenting classes to augmenting external generic function. This obviates the need for publicly accessible superclass method invocation entry points.[40]

---

39. “Matching” here means that the method’s tuple of specializers is a super type of the run-time type of the actual argument tuple in the implementation. We do not use the term “applicable” because, strictly speaking, such a method is not applicable to a superclass method invocation.
40. Other attempts to circumvent the encapsulation of external generic functions should not be possible given the privileged access used for the dispatcher objects. Only the `function` field of the external generic function is available, forcing clients to use the chain of responsibility in the proper order. Superclass method invocations that skip *local* methods of the chain do not cause encapsulation problems and have access to the private `old_function` field of the local, private dispatcher object.

a)
```
// compilation unit "record"
public void ActivityLog.recordAndCheckPoint( Transaction t ) {
    this.record(t);
    StaticCheckPoints.register(t);
}
public void ProtectedLog.recordAndCheckPoint( Transaction t ) {
    this.record(t);
    StaticCheckPoints.register(t);
    BackupCheckPoints.register(t);
}
```

---

b)
```
...
    public subvert( ActivityLog log, Transaction t ) {
        // breaks encapsulation when log is an instance of ProtectedLog
        recordAndCheckPoint$anchor.superEntryPointForProtectedLog.
            apply( log, t );
    }
```

Figure 47: Encapsulation problem with superclass method invocations from external methods: part a) shows an external generic function on the classes given in Figure 12, part b) shows a method that uses the proposed superclass method invocation entry points for self-augmenting classes to break the encapsulation of the `recordAndCheckPoint` generic function on a `ProtectedLog` instance.

6.1.3. Dispatcher Mates

The compilation-strategy induced restrictions on overriding methods could be relaxed (see the discussion of dispatcher mates on page 36). In regular Java an overriding method can grant more permissive privileged access than the method it overrides; for example, an overriding method of a protected method can be declared public. The current MultiJava restrictions allow this idiom between classes for internal methods of internal generic functions, providing backward compatibility with Java. But the idiom is not allowed between the multimethods a single class or between the methods of an external generic function. To support this idiom in a more general way requires a change in the compilation strategy. Instead of compiling a method into a dispatcher method with all of its dispatcher mates, the compiler could group methods based on their compatibility. For external generic functions a compiler would also have to add additional function fields to the anchor class and adjust the construction of the chain of responsibility to maintain encapsulation. Similar arguments apply to methods declared `final`.

Relaxing the restrictions on the exceptions thrown by overriding multimethods and methods of external generic functions to match the restrictions of regular Java is more straightforward. A single dispatcher method can still be used which declares that it may throw the union of the exceptions of all

the dispatcher mates. But this is not enough since it is possible for confusing error messages to occur with this strategy. For example, consider the following code permitted under the relaxed restrictions:

```
public void Log.writeTo( Writer wr ) throws IOException { /* ... */ }
public void Log.writeTo( StringWriter wr ) { /* ... */ }
```

A programmer using this generic function might reasonably expect the following code to compile without error:

```
public void showLog( Log l ) {
    StringWriter sWriter = new StringWriter();
    l.writeTo( sWriter );                          // line 3
    System.out.println( sWriter.toString() );
}
```

But with the compilation strategy given the target of the invocation in line 3 is the external generic functions `apply` method, which declares that it throws `IOException`. Thus without additional changes to the compilation strategy the compiler will report that in line 3 the exception `IOException` can be thrown and it is neither caught nor thrown by the enclosing method. The solution is to encode the exception information for each of the multimethods in the custom attributes of the generated class files. The compile-time method look-up procedure would have to be modified to not just identify the target generic function and most-specific (statically known) receiver class as in regular Java. Instead the most-specific (statically known) multimethod body would need to be identified.

Since MultiJava disallows abstract external methods and the abstract modifier on methods with explicit specializers, there is no need to relax the dispatcher mates restrictions for the abstract modifier. It may be interesting to investigate the semantics of an abstract method with explicit specializers, but we leave that and the corresponding dispatcher mates challenges for future work.

For the remaining method modifiers, `native`, `strictfp`, and `synchronized`, the dispatcher mates restrictions can simply be lifted. These are implementation specific modifiers and so a compilation strategy that ignores the modifiers when generating dispatcher methods but uses the modifiers on the *ident*`$body` multimethod bodies will have the correct semantics.

6.1.4. Other Forms of Augmentation

MultiJava currently allows only instance (non-`static`) methods to augment existing classes. However, it would be straightforward to extend the language to allow augmenting static methods and even augmenting static fields. Augmenting static methods and fields could be implemented via regular static members in anchor classes. A compiler could detect whether an invocation or field reference is to a regular static member or an augmenting member and insert bytecode to invoke or reference the

appropriate member. Clients of an augmenting static method or field would have static initializers that ensured the necessary anchor class was loaded and initialized.

Unlike augmenting static fields, augmenting instance fields would require more significant extensions to our compilation strategy. One can imagine using a hash table within an anchor class to store the values of the augmenting fields for each object. Ensuring that augmenting fields are initialized sensibly is problematic, particularly when an instance of the augmented class is created by a client that does not import the augmenting field. One solution, borrowed from subject-oriented programming, is to require that the declaration of an augmenting field includes a *deferred initializer*, a block of code that lazily initializes the field when it is first accessed [Harrison and Ossher 1993] (p. 417). This initializer might be some default initial constant value or might be calculated from the state of the augmented object. Related to the problem of initializing augmenting fields is that of serializing and deserializing [Arnold et al. 2000] (§15.7) the augmented classes. It may be that only transient augmenting fields can be supported with the deferred initializers used after deserializing objects.[41]

6.1.5. Specializing Other Parameters

MultiJava currently permits explicit specializers on only the formal parameters of method declarations. In Java formal parameters can appear in two other places, catch clauses and constructor declarations. There are no advantages to adding explicit specializers to the parameters in catch clauses, one can already achieve dynamic selection of catch clauses by polymorphism through simply ordering the caught exceptions from most to least specific.

It is easier to imagine applications of explicit specializers for constructors. For example:

```
public class SortedSelectionWidget {
    public SortedSelectionWidget(List items) {
        this.items = sort(items);
    }
    public SortedSelectionWidget(List@SortedList items) {
        this.items = items;
    }
    List items;
    /* ... */
}
```

This code fragment shows a class that implements a GUI selection widget with the items sorted alphabetically. If the list of items passed to the constructor is already sorted then the implementation need not sort the list again.

41. A field declared `transient` in Java is one whose value is lost (i.e., reset to its default) when the object is serialized and then deserialized [Gosling et al. 2000] (§8.3.1.3).

The challenge in allowing specialization on constructor parameters lies in the fact that constructors in Java are not methods and do not override constructors from their superclass, but they always invoke a superclass constructor, either implicitly or explicitly, before executing the remainder of their bodies [Gosling et al. 2000] (§8.8.5). Were we to add explicit specializers to the parameters of constructors it would be possible for one constructor to override another constructor in the same class as in the second `SortedSelectionWidget` constructor above. Such related constructors could be compiled into a single constructor that used typecases to select the appropriate constructor body. But these typecases would have to be executed *after* calling the superclass constructor. Thus a plausible restriction on overriding constructors is that any superclass constructor call appearing in the overridden constructor must have a matching superclass constructor call in the overriding constructor. The typecases would appear after a superclass constructor call in the single generated constructor.

On the other hand, it seems there is relatively little benefit in permitting specialization on constructor parameters. In the current incarnation of MultiJava one can already write factory methods [Gamma et al. 1995] (pp. 107–116) that use multiple dispatch. For example we could write the `SortedSelectionWidget` as:

```
public class SortedSelectionWidget {
    public static SortedSelectionWidget makeNew(List items) {
        return new SortedSelectionWidget(sort(items));
    }
    public static SortedSelectionWidget makeNew(List@SortedList items) {
        return new SortedSelectionWidget(items);
    }
    private SortedSelectionWidget(List items) {
        this.items = items;
    }
    List items;
    /* ... */
}
```

6.1.6. Null Specializers

One intriguing possibility is to modify the syntax and semantics of multimethods to permit the declaration of methods with `null` specializers. Were this done then the common coding pattern shown in this code fragment:

```
public void leftEdge( Shape s ) {
    if (s == null) {
        throw new IllegalArgumentException();
    } else {
        return leftMost(s.borderPoints());
    }
}
```

could be written

```
public void leftEdge( Shape s ) {
    return leftMost(s.borderPoints());
}
public void leftEdge( Shape@null n ) {
    throw new IllegalArgumentException();
}
```

Introducing `null` specializers would complexity to the implementation-side typechecking and compilation of multimethods. First, in order to prevent run-time ambiguities, the implementation-side typechecking would have to consider `null` to be among the visible types.[42] For example the following declarations would be ambiguous on the invocation `new Shape().closest(null,null)`:

```
public Shape Shape.closest(Shape s1, Shape s2) {
    return (s1.distanceTo(this) < s2.distanceTo(this)) ? s1 : s2;
}
public Shape Shape.closest(Shape@null s1, Shape s2) {
    return s2;
}
public Shape Shape.closest(Shape s1, Shape@null s2) {
    return s1;
}
```

The tuple subtyping relation (defined in Subsection 2.2.2 on page 26) would need to be extended to accommodate the `@null` explicit specializers. Also, since `instanceof` checks in the JVM always return false for a `null` argument, we would need to modify the implementation of dispatcher methods to check parameters for identity with `null`.

6.1.7. Link-time Checks

We could relax the typechecking restriction of Subsection 2.2.5 on page 30 by replacing the compile-time checks with link-time checks, as in Dubious's System E [Millstein and Chambers 1999]. This would increase the expressiveness of MultiJava while sacrificing some modularity. It would still, however, avoid dynamic checks for generic function ambiguity and incompleteness. We illustrate this with an example. Suppose the existence of a library that includes the following class:

```
// compilation unit "Employee"
package people;
public class Employee {
    /* ... */
}
```

42. The use of the two different subtyping operations, $\leq$: and $\leq_{mic}$, in the dispatch semantics prevents such ambiguities from arising in the current version of MultiJava despite not including `null` among the visible types for implementation-side typechecking. A method is applicable to an invocation with a `null` argument only if that method is not specialized in the `null` argument's position.

Next suppose that a new `benefits` package is developed, including the following external generic function:

```
// compilation unit "vacation"
package benefits;
import people.Employee;
public int Employee.vacation() {
    return baseDays() + daysPerYear() * yearsOfService();
}
// private methods act as constant fields
private int Employee.baseDays() {
    return 2;
}
private int Employee.daysPerYear() {
    return 5;
}
```

and another package is developed for tracking executive employees:

```
// compilation unit "Executive"
package bigwig;
import benefits.vacation;
import people.Employee;
public class Executive extends Employee {
    /* ... */
    // overrides the external method Employee.vacation()
    public int vacation() {
        return super.vacation() + EXEC_BONUS;
    }
    private static int EXEC_BONUS = 2;
}
```

So far this example does not run afoul of the typechecking restrictions. But suppose that another package is developed, independent of the `benefits` and `bigwig` packages. In this class is another `Employee` subclass:

```
// compilation unit "TempEmployee"
package serfs;
import people.Employee;
public class TempEmployee extends Employee {
    /* ... */
}
```

Finally, suppose an application arises that requires all of these classes and the `vacation` generic function to be used together. This is type sound; in particular invoking the `vacation` method on an instance of `TempEmployee` returns the value calculated by the `Employee.vacation` method declared in the `vacation` generic function. But suppose the client application wants the behavior implied by this declaration:

```
// compilation unit "vacation"
package client;
```

```
import benefits.vacation;
import serfs.TempEmployee;
// overrides the Employee.vacation method
public int TempEmployee.vacation() {   // disallowed by restriction R3!
    return 0;
}
```

This code attempts to add an external method to a non-local external generic function and is prohibited by restriction ***R3*** because such additions can sometimes lead to ambiguities that elude static detection (though here the new method does not).

By adding link-time checks MultiJava might permit an even greater variety of coding idioms. For example, relaxing these restrictions would allow MultiJava to solve the generalized binary method problem. Potential ambiguities would be detected at link-time, that is when the regular classes and anchor classes are initialized by the JVM. One would not have to wait until executing a problematic invocation to discover the ambiguity. One would be assured that run-time ambiguities could not arise, and so dynamic ambiguity checks would not be required.

The basic implementation strategy for these link-time checks is the addition of appropriate checking code to the static methods that construct the chain-of-responsibility for an external generic function. For internal multimethods a similar static method for link-time checks would need to be added. The exact content of these link-time checks remains as future work, but is indicated by the checks in System E of the Dubious language [Millstein and Chambers 1999].

Also, if R3 is lifted, it is not clear that there is a compilation strategy that permits dispatching for new external methods to be integrated with that for the existing internal methods (without non-modular recompilation of the existing classes). On the other hand, if the existing generic function were an external generic function it might be possible to manipulate, perhaps using dynamic compilation, the generic function's chain of responsibility to introduce the new external multimethods at the appropriate points in the dispatch sequence. This dynamic compilation corresponds to the link-time checks required for typechecking System E. If the dynamic compilation could be accomplished for external generic functions it might argue for compiling all generic functions using the chain-of-responsibility pattern. Or better still, a JVM could be designed that supports the separation of method hierarchies from class hierarchies and provides for dispatch over these method hierarchies in native code. We leave the investigation of these trade-offs in modularity and expressiveness as future work.

6.2. TupleJava

An early plan for adding multimethods to Java was to apply the concept of multiple dispatch as dispatch on tuples [Leavens and Millstein 1998], leading to TupleJava. In TupleJava, all multimethods would be external to classes. A multimethod that dispatched on 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 couples the method declaration and invocation code, forcing client code to change if the set of dispatched arguments changes. 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 single dispatch generic functions, 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 `Objects`. Thus, with TupleJava, the invocation in the code

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

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

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 advantage of TupleJava's congruence of syntax and semantics outweighs the expressiveness and code maintenance advantages of MultiJava. This investigation is left as future work.

SECTION 7. RELATED WORK

This section reviews the literature on previous approaches to solving the augmenting and binary method problems. It also highlights the solution to the modularity problem and some work on modular separation of concerns.

7.1. Augmenting Method Problem

Most previous work on the augmenting method problem has focused on the visitor pattern and its derivatives, discussed in Subsection 1.1.1. A notable recent idea that goes beyond clever extension of the visitor pattern is Zenger and Odersky's extensible datatypes with defaults [Zenger and Odersky 2001].

The authors describe a Java language extension that introduces extensible algebraic datatypes and a typecase construct. They describe a technique called "extensible algebraic datatypes with defaults" that is more powerful than the extensible visitor pattern and seems to solve the augmenting method problem. Extensible datatypes can be understood by analogy to MultiJava. A datatype along with its variants (for example, the datatype Term, with variants Variable, Lambda, and Application) is analogous to an abstract class and its concrete subclasses in MultiJava. Separate operations over datatypes, written using typecases, are analogous to MultiJava's external generic functions.

The first key idea is that operations over datatypes must provide a default case. This can be compared to the requirement in MultiJava that external methods may not be abstract. The authors cite empirical evidence that, in the domain of compiler construction, these default cases often provide the appropriate behavior for extensions of the datatypes, which is the second key idea. Datatypes may be extended through a mechanism that allows one to just specify the additional variants for the new datatype while inheriting the existing variants from the datatype being extended. In the MultiJava model one extends a datatype by declaring new concrete subclasses of the original abstract class.

When the default operation is not appropriate to a new variant, the author's language design permits extension of operations on datatypes. The extending operation can specify behavior for some of the new variants and defer to the extended operation in the other cases. One could model this in MultiJava by including the new behaviors as internal methods of the new variants. New variants that could use the default behavior would simply inherit the external default method. The disadvantage of this approach, versus extensible algebraic datatypes, is that the datatype and the operation become commingled in the new variants. One could avoid this by declaring external methods describing the behavior of the operation on the new variants. However, by restriction ***R3***, such external methods extending

a non-local generic function are prohibited. We have seen that, without ***R3***, unseen ambiguities and incompleteness can lead to run-time errors. Indeed, run-time errors are possible using the additional flexibility afforded under extensible algebraic datatypes [Zenger and Odersky 2001] (§4.3), though the authors report that such problems are rare in practice. The proposed link-time checks for MultiJava (Subsection 6.1.7 on page 97) would permit all the extensibility of the extensible algebraic datatypes approach while avoiding the potential for these sorts of run-time errors.

Another disadvantage with extensible algebraic datatypes is that the variants declared via the datatype construct cannot have their own behavior, encapsulated within the variants. That is, datatype variants contain only state information; they cannot contain internal methods. Thus, all operations on variants must be written externally. And since the variants must be declared separately, it is not possible to use the extensible algebraic datatypes mechanism to add operations to existing classes.

Seen in the context of MultiJava's goals, extensible algebraic datatypes solve the augmenting method problem, but only in situations where development proceeds from scratch. Extensible algebraic datatypes are a non-solution if development involves extension of a library for which source code is not available. Run-time errors that can occur with extensible algebraic datatypes are statically detected with MultiJava. And of course extensible algebraic datatypes have nothing to say about the binary method problem.

7.1.1. Modular Separation of Concerns

Subsuming the augmenting method problem is a concept that can be called "modular separation of concerns". *Modular separation of concerns* is the idea that a program should be structured so that code for common functionality (concerns, subjects, aspects) is grouped together instead of dispersed throughout a program [Parnas 1972, Parnas 1975]. Languages and programming environments that support modular separation of concerns typically allow, among other things, grouping of common operations on a hierarchy of classes into a single location, as in MultiJava's open classes. Subject-oriented programming is a very general manifestation of the modular separation of concerns concept. Before describing that we discuss a more specific manifestation, aspect-oriented programming.

Aspect-oriented programming [Kiczales et al. 1997], typified by the language AspectJ [Kiczales et al. 2001], provides support for modular separation of concerns via aspects. An aspect may specify additional code to be executed at "certain well-defined points in the execution of the program" [Kiczales et al. 2001] (p. 329) known as *join points*. This provides support for a sort of pattern-based meta-programming, allowing one to specify, for example, that a certain body of code should be executed whenever a method whose name begins with the string "open" is invoked. An aspect may also intro-

duce new methods to existing classes without modifying those classes, thus supporting open classes. However, aspects are not typechecked modularly. Instead, a whole-program analysis is required. MultiJava does not require whole-program analysis because its typechecking is modular. Like MultiJava, AspectJ targets the standard JVM; thus classes extended via aspects must be recompiled to implement the additional functionality. And although dependency analysis could be used to avoid whole program compilation, recompilation of classes extended via aspects is required if extending aspects are changed. MultiJava's open class technique does not require recompilation of the classes that are being extended. Unlike with AspectJ, in MultiJava a client of a library can add new methods to the classes of that library without polluting the interfaces of those classes (Subsection 2.1.7 on page 24). On the other hand, because it cannot edit the code of existing classes and because it does not have pattern-based metaprogramming, MultiJava cannot handle separation of concerns as well as AspectJ.

Analyzing AspectJ in light of the MultiJava's goals shows that, while AspectJ allows the addition of methods to classes that can be recompiled, it cannot be used to solve the augmenting method problem in situations where development extends a library for which bytecode must remain unchanged (e.g., due to other clients). Like extensible abstract datatypes, AspectJ does not address the binary method problem.

Harrison and Ossher describe a new programming paradigm that they call *subject-oriented programming* [Harrison and Ossher 1993]. Subject-oriented programming generalizes the object-oriented paradigm. A *subject* is roughly equivalent to an entire program in an object-oriented language in that all code within that subject shares the same set of class and type hierarchies, operations, and object state.[43] What makes subject-oriented programming unique is that disparate subjects, with distinct class and type hierarchies, operations, and object state, can share access to the same set of objects. This is akin to the idea in MultiJava that the apparent signature of a class depends on the external generic functions imported by the client. The only operation necessarily shared by subjects on a given object is the identity operation.

Various composition rules are used to combine subjects into programs. These rules specify mappings between class and type hierarchies in the composed subjects and describe how method dispatch from within one subject impacts the other subjects in the composition. For example, suppose several subjects each declared an operation with the same name and arguments for a given object. A composition rule might specify that an invocation of this operation in one subject should also execute the code for this operation in the other subjects. More complex composition rules can be imagined that map

43. This sharing is modulo privileged access restrictions.

between operations of different names and parameters and specify compositions of return types of the methods.

It is difficult to evaluate subject-oriented programming wholly in the context of MultiJava's goals. Subject-oriented programming is more a philosophy than an implementation technique or programming language. The underlying language in which the subjects are encoded is not fixed. Thus a subject-oriented programming language that used an underlying language providing multiple dispatch would solve the binary method problem. Subject-oriented programming cannot be used to extend code that was written in a regular object-oriented language. The original code must be written using the same subject-oriented programming language.

Subject-oriented programming's additional level of abstraction (beyond that of object-oriented programming), while expanding the expressiveness of the language, also brings the attendant expansion in complexity of reasoning. AspectJ and MultiJava can both be viewed as incremental approaches towards the more general subject-oriented philosophy. AspectJ maintains the central control structure of a single program, but allows additional operations and state to be in separate aspects. The dispatch flexibility of subject-oriented programming's composition rules is achieved through AspectJ's join points. To provide this flexibility AspectJ requires a whole program analysis. MultiJava's open classes allow additional operations to be specified via external generic functions while maintaining modular static typechecking and compilation. We are interested in investigating how much of the flexibility of subject-oriented programming can be achieved while still maintaining these desirable properties.

7.2. Adding Multiple Dispatch to Existing Single Dispatch Languages

Several proposals have been made for adding multiple dispatch to single dispatch languages. None of these proposals attempts to solve the augmenting method problem. They succeed in varying degrees at solving the binary method problem.

Encapsulated multimethods [Castagna 1995, Bruce et al. 1995] are a design for adding asymmetric multimethods to an existing single dispatch 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. While encap-

sulated multimethods can be viewed as a solution to the binary method problem, they involve tedious and error prone hand-coded dispatch.

Boyland and Castagna demonstrated the addition of asymmetric multimethods to Java using “parasitic methods” [Boyland and Castagna 1997]. 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.

Another approach to adding multiple dispatch to Java is given by Dutchyn, et al. [Dutchyn et al. 2001]. In that work the authors use a *marker interface*, an empty interface that marks implementing classes. A modified JVM detects the marker and changes the semantics of Java’s static overloading for the marked classes. Methods that would be considered statically overloaded in regular Java are instead treated as overriding methods of the same generic function. The paper presents benchmarking results that compare regular Java code using the double-dispatching technique versus a multiple dispatch version of the same code running on the modified JVM. The results indicate only a minor slowdown (3 to 5%) for code that does not use multiple dispatch and a speedup of 1.21 for multiple dispatch code that replaces code using the double-dispatching technique.

There are, however, several drawbacks to the given approach to multiple dispatch in Java. Because no additional typechecking restrictions are applied, Java’s modular, static typechecking is lost; a whole-program analysis is required to detect potential multimethod ambiguities. Such an analysis may not even be possible, for example when proprietary classes from separate sources are introduced into a long-running server environment. Because of this possibility, the modified JVM must perform run-time checks for ambiguities, throwing an exception if an ambiguity is detected.

Because of the marker interface technique, the granularity of application for multiple dispatch is quite coarse; the marker interface is applied at the class level and all methods of the marked class and its subclasses are dispatched according to the new multiple dispatch semantics. Also all parameters of

each such method are considered in making dispatch decisions. Since not all methods benefit from multiple dispatch, this coarse granularity could potentially result in additional run-time costs for methods that do not benefit from multiple dispatch.[44]

Although the modified JVM technique solves the binary method problem by supporting multiple dispatch, it does so it a way that does not respect the semantics of existing Java code using static overloading. Also, since the semantics are changed by changing the JVM the semantics of marked classes are different if they are executed by a standard JVM. Thus the modified JVM technique fails to satisfy three of the four constraints, given in Subsection 1.2, for our solution.

Because of its modular, static typechecking and fine-grained dispatch specification, the MultiJava language overcomes the shortcomings of the marker interface approach. An interesting avenue for future work is to consider development of a custom JVM, akin to that developed by Dutchyn, et al., to improve the run-time efficiency of code that uses multiple dispatch (see Subsection 3.5 on page 61).

7.3. Modularity Problem

In the design of the Dubious language Millstein and Chambers present a solution to the modularity problem for multimethods and open classes [Millstein and Chambers 1999]. Dubious is a simple core language based on multimethods and open classes. The authors describe 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.

44. An unpublished extension to this work uses the custom attribute approach in bytecode to improve the granularity of multiple dispatch (personal communication with Christopher Dutchyn, June 2001).

SECTION 8. CONCLUSIONS

We conclude by examining several promising avenues for future work building on MultiJava and by summarizing the contributions of this research.

8.1. Future Work

One area of future work is the implementation of some of the various language extensions suggested in Subsection 6.1 on page 91. The most interesting of these extensions are the addition of link-time checks to relax the typechecking restrictions, the addition of other kinds of augmenting members, like augmenting fields, and the addition of `null` specializers. It seems that these would do the most to increase the expressiveness of the language. Allowing specializers to disambiguate overridden method invocations and relaxing the implementation-strategy induced restrictions on dispatch mates will be considered if experience indicates that these features are useful. On the other hand, it seems unlikely that allowing self-augmenting classes and the specialization of parameters in constructors will be needed. Another implementation task that remains open is that of creating a custom JVM for MultiJava to improve dispatch efficiency.

The extensibility of MultiJava makes it an excellent language for developing compilers. This investigation will begin with translating of sections of `mjc` itself into MultiJava. We are also interested in a MultiJava compiler built from the ground up in MultiJava.

In the general area of research on multiple dispatch we are interested in investigating the notion of behavioral subtyping and formal specification of multimethods. Work is under way to build a new JML typechecker atop `mjc`. This will provide the platform for such an investigation.

Another area of future work focuses on the programmer's conceptual model of multiple dispatch. It is the author's belief that a large part of the wide-spread acceptance of object-oriented languages is the simple conceptual model presented to the programmer. As we like to put it, people see the world in nouns. Objects allow the program to model a system as a set of interacting nouns.

Certain operations make sense in this object-centric view. For example, it makes sense for a horse to be responsible for the action of eating an apple. However, other operations, including many binary methods, do not fit well in this world view. For example, it does not make sense for one horse to be responsible for the action of two horses racing. Conceptually both horses are equal partners (or competitors) in the race; one does not take the other on its back and run the race alone. Less facetiously, causing a group of software agents to negotiate is another example where it may not make sense to conceive of just one member of the group to be taking action in the negotiation. The TupleJava syntax

reflects this more symmetric concept of the objects participating in an action. Future work includes investigating whether the sacrifices in extensibility (discussed in Subsection 6.2) are offset by this supposed conceptual advantage. Initiating this investigation should not be difficult since it just involves modifying the mjc parser to process and desugar TupleJava syntax.

Also related to the programmer's conceptual model of multimethods is the development of tools for viewing, creating, and manipulating programs that use multimethods. Part of this is developing a better understanding of multimethod hierarchies as distinct structures from class hierarchies. One idea is the development of a viewer that displays the lattice formed by argument tuples and the tuple subtype relation. Superimposed on this lattice would be symbols indicating the implemented multimethods. Virtual reality technology might be useful in rendering such a display.

In the area of research on modular program extension, we are interesting in investigating how much of the promise of subject-oriented programming can be achieved in a language with modular, static typechecking and compilation. For example, which of AspectJ's join points can be supported by allowing clients to import aspects, as they import external generic functions in MultiJava, without actually modifying the code augmented by those aspects. Adding augmenting fields as discussed above also contributes towards this subject-oriented programming ideal. Since then different clients of an object can share the intrinsic state of an object while maintaining separate subject-based extrinsic state.

Java provides an extensive reflection mechanism [Arnold et al. 2000] (§11.2). Of course there is no provision in this mechanism for determining what external generic functions are in the apparent signature of a class from the perspective of a client. However, in a nice recursion, using MultiJava's open classes we might augment `java.lang.Class` to provide just this functionality.

Finally, a compiler is never complete. We look forward to developing a user community and having its feedback guide the development and refinement of mjc.

8.2. Contributions

We have described the design, semantics, typechecking, and compilation of MultiJava. While much of this appeared in CLCM2000, we have expanded the treatment both in depth and breadth. Additionally we have presented an implementation of mjc, a compiler for the MultiJava language. We have also demonstrated the utility of the language through the implementation of an extending series of interpreters, and the efficiency of the language through empirical testing.

MultiJava solves the augmenting method and binary method problems in situations where development extends existing libraries or proceeds from scratch. It does so while maintaining modular edit-

ing, typechecking, and compilation, making it the first language to do so. By targeting the standard JVM and generating bytecode as efficient as any other modular solution to these problems, MultiJava is suitable for production software development as well as research work. The design of MultiJava is carefully “Java-like” so that experienced Java developers should have little difficulty in adopting the language.

Additional specific contributions of this paper include:

- a discussion of pleomorphic methods and their compilation,
- increasing the expressiveness of MultiJava by adding support for run-time dispatching on array types and overloading of external generic functions,
- additional restrictions on the use of some language features that, while not necessary for soundness, provide software engineering benefits,
- a thorough treatment of upcalls that identifies and remedies an encapsulation problem not previously discussed and a compilation strategy for upcalls that handles all legal combinations of sender and target methods,
- a compilation strategy for private methods written using the open class syntax, and
- a technique for encoding MultiJava specific information in bytecode for efficient retrieval by MultiJava compilers.

We look forward to fostering and participating in the evolution of MultiJava and invite others to join us at `http://www.multijava.org`.

APPENDIX A. PERFORMANCE EVALUATION CODE

This appendix gives the source code used to measure the performance results discussed in Section 5. The appendix is divided into three subsections giving the source code for the two sorts of tests (open classes and multimethods) and the raw data from running those tests.

A.1. Source Code for Open Class Tests

```
// compilation unit Tree.java
package evaluation.speed;

public class Tree {
    public Tree( Object value ) {
        this.value = value;
    }

    // internal implementation of prettyPrinting to measure regular
    // dispatch speed
    public String internalPrettyPrint() {
        return internalPrettyPrint( "" );
    }

    public String internalPrettyPrint( String prefix ) {
        return prefix + value() + "\n";
    }

    public Object value() {
        return value;
    }

    private Object value;
}
```

```
// compilation unit Interior.java
package evaluation.speed;

public class Interior extends Tree {
    public Interior( Object value, Tree left, Tree right ) {
        super( value );
        this.left = left;
        this.right = right;
    }

    // internal implementation of prettyPrinting to measure regular
    // dispatch speed
    public String internalPrettyPrint( String prefix ) {
        StringBuffer result =
            new StringBuffer( prefix + value() + "\n" );

        String newPrefix = prefix + "|  ";
        result.append( left().internalPrettyPrint( newPrefix ) );
        result.append( right().internalPrettyPrint( newPrefix ) );
```

```
        return result.toString();
    }

    public Tree left() {
        return left;
    }

    public Tree right() {
        return right;
    }

    private Tree left;
    private Tree right;
}
```

```
// compilation unit dispatchTest.java
package evaluation.speed;

include evaluation.speed.dispatchTest;

public void Tree.dispatchTest() {
}

public void Interior.dispatchTest() {
    left().dispatchTest();
    right().dispatchTest();
}
```

```
// compilation unit size.java
package evaluation.speed;

include evaluation.speed.size;

public int Tree.size() {
    return 1;
}

public int Interior.size() {
    return 1 + left().size() + right().size();
}
```

```
// compilation unit prettyPrint.java
package evaluation.speed;

include evaluation.speed.prettyPrint;

public String Tree.prettyPrint() {
    return prettyPrint( "" );
}
```

```
public String Tree.prettyPrint( String prefix ) {
    return prefix + value() + "\n";
}

public String Interior.prettyPrint( String prefix ) {
    // We would use overriddenMethod(prefix) in place of the argument
    // in the following if that feature were implemented.
    StringBuffer result = new StringBuffer( prefix + value() + "\n" );

    String newPrefix = prefix + "|  ";
    result.append( left().prettyPrint( newPrefix ) );
    result.append( right().prettyPrint( newPrefix ) );
    return result.toString();
}
```

// compilation unit MultiInterior.java

```
package evaluation.speed;

include evaluation.speed.prettyPrint;
include evaluation.speed.dispatchTest;
include evaluation.speed.size;

public class MultiInterior extends Tree {
    public MultiInterior( Object value, Tree[] children ) {
        super( value );
        this.children = children;
    }

    public String prettyPrint( String prefix ) {
        // We would use overriddenMethod(prefix) in place of the argument
        // in the following if that feature were implemented.
        StringBuffer result = new StringBuffer( prefix + value() + "\n" );

        String newPrefix = prefix + "|  ";
        for (int i = 0; i < children.length; i++) {
            result.append( children[i].prettyPrint( newPrefix ) );
        } // end of for
        return result.toString();
    }

    public void dispatchTest() {
        for (int i = 0; i < children.length; i++) {
            children[i].dispatchTest();
        } // end of for
    }

    public int size() {
        int result = 1;
        for (int i = 0; i < children.length; i++) {
            result += children[i].size();
        } // end of for
        return result;
    }
```

```
    public Tree[] children() {
        return children;
    }

    private Tree[] children;
}
```

```
// compilation unit VTree.java
package evaluation.speed;

public class VTree {
    public VTree( Object value ) {
        this.value = value;
    }

    public void accept( VTreeVisitor v ) {
        v.visitVTree( this );
    }

    public Object value() {
        return value;
    }

    private Object value;
}
```

```
// compilation unit VInterior.java
package evaluation.speed;

public class VInterior extends VTree {
    public VInterior( Object value, VTree left, VTree right ) {
        super( value );
        this.left = left;
        this.right = right;
    }

    public void accept( VTreeVisitor v ) {
        v.visitVInterior( this );
    }

    public VTree left() {
        return left;
    }

    public VTree right() {
        return right;
    }

    private VTree left;
    private VTree right;
}
```

```
// compilation unit VTreeVisitor.java
package evaluation.speed;

public interface VTreeVisitor {
    VTreeVisitor makeInstance();
    String name();
    void visitVTree( VTree tree );
    void visitVInterior( VInterior interior );
}
```

```
// compilation unit VTreeDispatchTester.java
package evaluation.speed;

public class VTreeDispatchTester implements VTreeVisitor {
    public VTreeDispatchTester() { }

    public VTreeVisitor makeInstance() {
        return new VTreeDispatchTester();
    }

    public void visitVTree( VTree tree ) {
    }

    public void visitVInterior( VInterior interior ) {
        interior.left().accept( this );
        interior.right().accept( this );
    }

    public String name() {
        return "dispatch tester";
    }
}
```

```
// compilation unit VTreeSizer.java
package evaluation.speed;

public class VTreeSizer implements VTreeVisitor {
    public VTreeSizer() {
        size = 0;
    }

    // factory method
    public VTreeVisitor makeInstance() {
        return new VTreeSizer();
    }

    public void visitVTree( VTree tree ) {
        size = 1;
    }

    public void visitVInterior( VInterior interior ) {
        VTreeSizer lSizer = (VTreeSizer) makeInstance();
        VTreeSizer rSizer = (VTreeSizer) makeInstance();
```

```
        interior.left().accept( lSizer );
        interior.right().accept( rSizer );
        size = 1 + lSizer.result() + rSizer.result();
    }

    public String name() {
        return "sizer";
    }

    public int result() {
        return size;
    }

    protected int size;
}
```

// compilation unit VTreePrettyPrinter.java

```
package evaluation.speed;

public class VTreePrettyPrinter implements VTreeVisitor {
    public VTreePrettyPrinter() {
        this( "" );
    }

    public VTreePrettyPrinter( String prefix ) {
        this.prefix = prefix;
    }

    public VTreeVisitor makeInstance() {
        return new VTreePrettyPrinter();
    }

    public void visitVTree( VTree tree ) {
        result.append( prefix + tree.value() + "\n" );
    }

    public void visitVInterior( VInterior interior ) {
        // Could also use visitVTree(interior) in place of the following:
        result.append( prefix + interior.value() + "\n" );

        VTreePrettyPrinter pp = (VTreePrettyPrinter) makeInstance();
        pp.prefix = prefix + "|  ";
        interior.left().accept( pp );
        result.append( pp.result() );

        pp = (VTreePrettyPrinter) makeInstance();
        pp.prefix = prefix + "|  ";
        interior.right().accept( pp );
        result.append( pp.result() );
    }

    public String result() {
        return result.toString();
```

```
    }

    public String name() {
        return "pretty-printer";
    }

    protected String prefix;

    protected StringBuffer result = new StringBuffer( "" );
}
```

// compilation unit VMultiInterior.java

```
package evaluation.speed;

public class VMultiInterior extends VTree {
    public VMultiInterior( Object value, VTree[] children ) {
        super( value );
        this.children = children;
    }

    public void accept( VTreeVisitor vis ) {
        // run-time ClassCastException if wrong type of visitor is used
        ((VMultiTreeVisitor) vis).visitVMultiInterior( this );
    }

    public VTree[] children() {
        return children;
    }

    private VTree[] children;
}
```

// compilation unit VMultiTreeVisitor.java

```
package evaluation.speed;

public interface VMultiTreeVisitor extends VTreeVisitor {
    void visitVMultiInterior( VMultiInterior interior );
}
```

// compilation unit VMultiTreeDispatchTester.java

```
package evaluation.speed;

public class VMultiTreeDispatchTester extends VTreeDispatchTester
    implements VMultiTreeVisitor
{
    public VTreeVisitor makeInstance() {
        return new VMultiTreeDispatchTester();
    }

    public void visitVMultiInterior( VMultiInterior interior ) {
        VTree[] myChildren = interior.children();
        for (int i = 0; i < myChildren.length; i++) {
```

```
            myChildren[i].accept( this );
        } // end of for
    }
}
```

// compilation unit VMultiTreeSizer.java

```
package evaluation.speed;

public class VMultiTreeSizer extends VTreeSizer
    implements VMultiTreeVisitor
{
    public VMultiTreeSizer() {
        super();
    }

    public VTreeVisitor makeInstance() {
        return new VMultiTreeSizer();
    }

    public void visitVMultiInterior( VMultiInterior interior ) {
        VTree[] children = interior.children();
        size = 1;
        for (int i = 0; i < children.length; i++) {
            VTreeSizer sizer = (VTreeSizer) makeInstance();
            children[i].accept(sizer);
            size += sizer.result();
        } // end of for
    }
}
```

// compilation unit VMultiTreePrettyPrinter.java

```
package evaluation.speed;

public class VMultiTreePrettyPrinter extends VTreePrettyPrinter
    implements VMultiTreeVisitor
{
    public VMultiTreePrettyPrinter() {
        super();
    }

    public VMultiTreePrettyPrinter( String prefix ) {
        super( prefix );
    }

    public VTreeVisitor makeInstance() {
        return new VMultiTreePrettyPrinter();
    }

    public void visitVMultiInterior( VMultiInterior interior ) {
        // Could also use visitVTree(interior) in place of the following:
        result.append( prefix + interior.value() + "\n" );

        VTreePrettyPrinter pp;
```

```
        String newPrefix = prefix + "|  ";
        VTree[] children = interior.children();
        for (int i = 0; i < children.length; i++) {
            pp = (VTreePrettyPrinter) makeInstance();
            pp.prefix = newPrefix;
            children[i].accept( pp );
            result.append( pp.result() );
        }
    }
}
```

```
// compilation unit VTreeNonExtensiblePrettyPrinter.java
package evaluation.speed;

public class VTreeNonExtensiblePrettyPrinter implements VTreeVisitor {
    public VTreeNonExtensiblePrettyPrinter() {
        this( "" );
    }

    public VTreeNonExtensiblePrettyPrinter( String prefix ) {
        this.prefix = prefix;
    }

    // required to satisfy interface, not used in algorithm
    public VTreeVisitor makeInstance() {
        return new VTreeNonExtensiblePrettyPrinter();
    }

    public void visitVTree( VTree tree ) {
        result.append( prefix + tree.value() + "\n" );
    }

    public void visitVInterior( VInterior interior ) {
        result.append( prefix + interior.value() + "\n" );

        String newPrefix = prefix + "|  ";
        VTreePrettyPrinter pp = new VTreePrettyPrinter( newPrefix );
        interior.left().accept( pp );
        result.append( pp.result() );

        pp = new VTreePrettyPrinter( newPrefix );
        interior.right().accept( pp );
        result.append( pp.result() );
    }

    public String result() {
        return result.toString();
    }

    public String name() {
        return "pretty-printer";
    }

    protected String prefix;
```

```
    protected StringBuffer result = new StringBuffer( "" );
}
```

```
// compilation unit Exercise.java
package evaluation.speed;

include evaluation.speed.prettyPrint;
include evaluation.speed.size;

public class Exercise {
    public static void main (String[] args) {
        System.out.println();
        System.out.println("binary tree using regular Java methods:");
        System.out.println("N = " + binaryTree.size() );
        System.out.println( binaryTree.internalPrettyPrint() );

        System.out.println();
        System.out.println("binary tree using open classes:");
        System.out.println("N = " + binaryTree.size() );
        System.out.println( binaryTree.prettyPrint() );

        System.out.println();
        System.out.println("n-ary tree using open classes:");
        System.out.println("N = " + naryTree.size() );
        System.out.println( naryTree.prettyPrint() );

        System.out.println();
        System.out.println("big n-ary tree using open classes:");
        System.out.println("N = " + bigNaryTree.size() );
        System.out.println( bigNaryTree.prettyPrint() );

        System.out.println();
        System.out.println("binary tree using regular visitor pattern:");
        System.out.println("N = " + binaryTree.size() );
        VTreeNonExtensiblePrettyPrinter regPP =
            new VTreeNonExtensiblePrettyPrinter();
        binaryVTree.accept( regPP );
        System.out.println( regPP.result());

        System.out.println();
        System.out.println("binary tree using extensible visitor pattern:");
        VTreeSizer sizer = new VTreeSizer();
        binaryVTree.accept( sizer );
        System.out.println("N = " + sizer.result());
        VTreePrettyPrinter vis = new VTreePrettyPrinter();
        binaryVTree.accept( vis );
        System.out.println(vis.result());

        System.out.println();
        System.out.println("n-ary tree using extensible visitor pattern:");
        VMultiTreeSizer mSizer = new VMultiTreeSizer();
        naryVTree.accept( mSizer );
        System.out.println("N = " + mSizer.result());
```

```
        VMultiTreePrettyPrinter mVis = new VMultiTreePrettyPrinter();
        naryVTree.accept( mVis );
        System.out.println(mVis.result());

        System.out.println();
        System.out.println("big n-ary tree using extensible visitor pat-
tern:");
        mSizer = new VMultiTreeSizer();
        bigNaryVTree.accept( mSizer );
        System.out.println("N = " + mSizer.result());
        mVis = new VMultiTreePrettyPrinter();
        bigNaryVTree.accept( mVis );
        System.out.println(mVis.result());
    } // end of main ()

    public static final Tree binaryTree =
          new Interior( "Object",
                      new Tree( "String" ),
                      new Interior( "Number",
                                  new Tree( "Integer" ),
                                  new Tree( "Float" )));

    public static final Tree naryTree =
        new MultiInterior( "Object",
                           new Tree[] {
                               new Tree( "String" ),
                               new MultiInterior( "Number",
                                                  new Tree[] {
                                                      new Tree( "Integer" ),
                                                      new Tree( "Float" ),
                                                      new Tree( "Long" ),
                                                  } ),
                               new Tree( "Class" ),
                           } );

    public static final Tree bigNaryTree;
    static {
        Tree[] bottom = new Tree[B];
        for (int i = 0; i < B; i++) {
            bottom[i] = new Tree( new Integer(i) );
        }

        for (int depth = 0; depth < D-1; depth++) {
            Tree[] next = new Tree[B];
            for (int i = 0; i < B; i++) {
                next[i] = new MultiInterior( new Integer(i), bottom );
            }
            bottom = next;
        }
        bigNaryTree = new MultiInterior( "root", bottom );
    }


    public static final VTree binaryVTree =
          new VInterior( "Object",
```

```
                        new VTree( "String" ),
                        new VInterior( "Number",
                                       new VTree( "Integer" ),
                                       new VTree( "Float" )));

    public static final VTree naryVTree =
        new VMultiInterior( "Object",
                            new VTree[] {
                                new VTree( "String" ),
                                new VMultiInterior( "Number",
                                                    new VTree[] {
                                                        new VTree( "Integer"
),
                                                        new VTree( "Float" ),
                                                        new VTree( "Long" ),
                                                    } ),
                                new VTree( "Class" ),
                            } );


    public static final VTree bigNaryVTree;
    static {
        VTree[] bottom = new VTree[B];
        for (int i = 0; i < B; i++) {
            bottom[i] = new VTree( new Integer(i) );
        }

        for (int depth = 0; depth < D-1; depth++) {
            VTree[] next = new VTree[B];
            for (int i = 0; i < B; i++) {
                next[i] = new VMultiInterior( new Integer(i), bottom );
            }
            bottom = next;
        }
        bigNaryVTree = new VMultiInterior( "root", bottom );
    }

    public static final int B = 4;
    public static final int D = 4;

}
```

```
// compilation unit TestOpenClassDispatch.java
package evaluation.speed;

import java.util.Date;
include evaluation.speed.prettyPrint;
include evaluation.speed.dispatchTest;
include evaluation.speed.size;

public class TestOpenClassDispatch extends Exercise {
    public static void main (String[] args) {
        if (args.length > 0) {
            ITERATIONS = Integer.parseInt( args[0] );
```

```
        }

        long startTime;
        long endTime;

        startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
        }
        endTime = new Date().getTime();
        System.out.println(ITERATIONS + " iterations of nothing: " +
                           (endTime - startTime) + " ms");

        testOpenClassDispatch( binaryTree );
        testOpenClassDispatch( naryTree );
        testOpenClassDispatch( bigNaryTree );

        testOpenClassPrettyPrint( binaryTree );
        testOpenClassPrettyPrint( naryTree );
        testOpenClassPrettyPrint( bigNaryTree );

        testOpenClassSize( binaryTree );
        testOpenClassSize( naryTree );
        testOpenClassSize( bigNaryTree );

        testVisitorDispatcher( binaryVTree, false );
        testVisitorDispatcher( naryVTree, true );
        testVisitorDispatcher( bigNaryVTree, true );

        testVisitorPrettyPrinter( binaryVTree, false );
        testVisitorPrettyPrinter( naryVTree, true );
        testVisitorPrettyPrinter( bigNaryVTree, true );

        testVisitorSizer( binaryVTree, false );
        testVisitorSizer( naryVTree, true );
        testVisitorSizer( bigNaryVTree, true );

        testRegularPrettyPrint( binaryTree );
        testNEVisitorPrettyPrinter( binaryVTree );

    }

    public static void testOpenClassDispatch(Tree t) {
        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            t.dispatchTest();
        } // end of for
        long endTime = new Date().getTime();
        System.out.println(ITERATIONS +
                           " iterations of external method dispatchTest: N = "
                           + t.size() + ", " + (endTime - startTime) + " ms"
);
    }

    public static void testOpenClassPrettyPrint(Tree t) {
        String result;
```

```
        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            result = t.prettyPrint();
        } // end of for
        long endTime = new Date().getTime();
        System.out.println(ITERATIONS +
                           " iterations of external method prettyPrint: N = "
                           + t.size() + ", " + (endTime - startTime) + " ms"
);
    }

    public static void testOpenClassSize(Tree t) {
        int result;
        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            result = t.size();
        } // end of for
        long endTime = new Date().getTime();
        System.out.println(ITERATIONS +
                           " iterations of external method size: N = "
                           + t.size() + ", " + (endTime - startTime) + " ms"
);
    }

    public static void testVisitorDispatcher(VTree t, boolean isMulti) {
        VTreeVisitor proto =
            isMulti ?
            new VMultiTreeDispatchTester() : new VTreeDispatchTester();

        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            VTreeVisitor vis = proto.makeInstance();
            t.accept( vis );
        } // end of for
        long endTime = new Date().getTime();

        VTreeSizer sizer = new VMultiTreeSizer();
        t.accept(sizer);
        System.out.println(ITERATIONS +
                           " iterations of visitor " + proto.name() + ": N = "
                           + sizer.result() + ", " +
                           (endTime - startTime) + " ms" );
    }

    public static void testVisitorPrettyPrinter(VTree t, boolean isMulti) {
        VTreeVisitor proto =
            isMulti ? new VMultiTreePrettyPrinter() : new VTreePrettyP-
rinter();

        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            VTreePrettyPrinter vis = (VTreePrettyPrinter) proto.makeIn-
stance();
            t.accept( vis );
            String result = vis.result();
```

```
        } // end of for
        long endTime = new Date().getTime();

        VTreeSizer sizer = new VMultiTreeSizer();
        t.accept(sizer);
        System.out.println(ITERATIONS +
                           " iterations of visitor " + proto.name() + ": N = "
                           + sizer.result() + ", " +
                           (endTime - startTime) + " ms" );
    }

    public static void testVisitorSizer(VTree t, boolean isMulti) {
        VTreeVisitor proto =
            isMulti ? new VMultiTreeSizer() : new VTreeSizer();

        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            VTreeSizer vis = (VTreeSizer) proto.makeInstance();
            t.accept( vis );
            int result = vis.result();
        } // end of for
        long endTime = new Date().getTime();

        VTreeSizer sizer = new VMultiTreeSizer();
        t.accept(sizer);
        System.out.println(ITERATIONS +
                           " iterations of visitor " + proto.name() + ": N = "
                           + sizer.result() + ", " +
                           (endTime - startTime) + " ms" );
    }

    public static void testRegularPrettyPrint(Tree t) {
        String result;
        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            result = t.internalPrettyPrint();
        } // end of for
        long endTime = new Date().getTime();
        System.out.println(ITERATIONS +
                           " iterations of regular method internalPrettyP-
rint:"
                           + " N = " + t.size() + ", " +
                           (endTime - startTime) + " ms" );
    }

    public static void testNEVisitorPrettyPrinter(VTree t) {
        VTreeNonExtensiblePrettyPrinter vis;
        String result;
        long startTime = new Date().getTime();
        for (int i = 0; i < ITERATIONS; i++) {
            vis = new VTreeNonExtensiblePrettyPrinter();
            t.accept( vis );
            result = vis.result();
        } // end of for
        long endTime = new Date().getTime();
```

```
        VTreeSizer sizer = new VMultiTreeSizer();
        t.accept(sizer);
        System.out.println(ITERATIONS +
                           " iterations of non-extensible pretty-print " +
                           "visitor: N = " + sizer.result() + ", " +
                           (endTime - startTime) + " ms" );
    }

    public static int ITERATIONS = 1000;
}
```

A.2. Source Code for Multiple Dispatch Tests

```
// compilation unit Real.java
package evaluation.speed2;

public class Real {
    public Real( double value ) {
        this.value = value;
    }

    // Generic function using multiple dispatch
    public Real multiply1( Real other ) {
        return new Real( value * other.value() );
    }

    // Generic function using typecases
    public Real multiply2( Real other ) {
        return new Real( value * other.value() );
    }

    // Generic function using double dispatch
    public Real multiply3( Real other ) {
        return other.multiply3Real( this );
    }

    public Real multiply3Real( Real other ) {
        return new Real( value * other.value() );
    }

    public Real multiply3Integer( Integer other ) {
        return new Real( value * other.value() );
    }

    public Real multiply3Rational( Rational other ) {
        return new Real( value * other.value() );
    }

    public double value() {
        return value;
    }

    public String toString() {
```

```
        return "" + value;
    }

    private double value;
}
```

// compilation unit Integer.java

```
package evaluation.speed2;

public class Integer extends Real {
    public Integer( long lValue ) {
        super( lValue );
        this.lValue = lValue;
    }

    // Generic function using multiple dispatch
    public Real multiply1( Real@Integer other ) {
        return new Integer( lValue * other.lValue() );
    }

    public Real multiply1( Real@Rational other ) {
        return new Rational( lValue * other.numerator(),
                             other.denominator() );
    }

    // Generic function using typecases
    public Real multiply2( Real other ) {
        if (other instanceof Integer) {
            return new Integer( lValue * ((Integer) other).lValue() );
        } else if (other instanceof Rational) {
            return new Rational( lValue * ((Rational) other).numerator(),
                                 ((Rational) other).denominator() );
        } else {
            return super.multiply2(other);
        }
    }

    // Generic function using double dispatch
    public Real multiply3( Real other ) {
        return other.multiply3Integer( this );
    }

    public Real multiply3Integer( Integer other ) {
        return new Integer( lValue * other.lValue() );
    }

    public Real multiply3Rational( Rational other ) {
        return new Rational( lValue * other.numerator(), other.denominator()
);
    }

    public long lValue() {
        return lValue;
    }
```

```
    public String toString() {
        return "" + lValue;
    }

    private long lValue;
}
```

// compilation unit Rational.java

```
package evaluation.speed2;

public class Rational extends Real {
    public Rational( long numerator, long denominator ) {
        super( ((double) numerator) / ((double) denominator) );
        this.numerator = numerator;
        this.denominator = denominator;
    }

    // Generic function using multiple dispatch
    public Real multiply1( Real@Integer other ) {
        return new Rational( numerator * other.lValue(), denominator );
    }

    public Real multiply1( Real@Rational other ) {
        return new Rational( numerator * other.numerator(),
                             denominator * other.denominator() );
    }

    // Generic function using typecases
    public Real multiply2( Real other ) {
        if (other instanceof Integer) {
            return new Rational( numerator * ((Integer) other).lValue(),
                                 denominator );
        } else if (other instanceof Rational) {
            return new Rational(numerator * ((Rational) other).numerator(),
                            denominator *((Rational) other).denominator());
        } else {
            return super.multiply2(other);
        }
    }

    // Generic function using double dispatch
    public Real multiply3( Real other ) {
        return other.multiply3Rational( this );
    }

    public Real multiply3Integer( Integer other ) {
        return new Rational( numerator * other.lValue(), denominator );
    }

    public Real multiply3Rational( Rational other ) {
        return new Rational( numerator * other.numerator(),
                             denominator * other.denominator() );
```

```
    }

    public long numerator() {
        return numerator;
    }

    public long denominator() {
        return denominator;
    }

    public String toString() {
        return numerator + "/" + denominator;
    }

    private long numerator;
    private long denominator;
}
```

// compilation unit Exercise.java

```
package evaluation.speed2;

public class Exercise {
    public static void main (String[] args) {
        for (int i = 0; i < names.length; i++) {
            for (int j = 0; j < names.length; j++) {
                test( names[i], names[j], values[i], values[j] );
            }
        }

    }

    public static void test( String xName, String yName, Real x, Real y ) {
        Real result1 = x.multiply1(y);
        Real result2 = x.multiply2(y);
        Real result3 = x.multiply3(y);
        System.out.println(xName + " * " + yName + " = " +
                           result1 + "\t" + result2 + "\t" + result3 );
    }

    public final static String[] names =
        new String[] { "pi", "two", "third" };
    public final static Real[] values =
        new Real[] {
            new Real( 3.14159265358979323846264338 3 ),
            new Integer( 2 ),
            new Rational( 1, 3 ),
        };

    static {
        if (names.length != values.length) {
            throw new RuntimeException("array lengths mismatched in " +
                                       "initializers" );
        } // end of if
    }
```

```
}
```

// compilation unit TestMultipleDispatch.java

```
package evaluation.speed2;
import java.util.Date;

public class TestMultipleDispatch extends Exercise {
    public static void main (String[] args) {
        if (args.length > 0) {
            ITERATIONS = java.lang.Integer.parseInt( args[0] );
        }

        testNothing();
        testMultiply1();
        testMultiply2();
        testMultiply3();
    }

    public static void testNothing() {
        System.out.print( ITERATIONS + " iterations of doing nothing: " );
        long startTime = new Date().getTime();
        for (int iter = 0; iter < ITERATIONS; iter++ ) {
            for (int i = 0; i < names.length; i++) {
                for (int j = 0; j < names.length; j++) {
                }
            }
        }
        long endTime = new Date().getTime();
        System.out.println((endTime - startTime) + " ms");
    }

    public static void testMultiply1() {
        System.out.print( ITERATIONS + " iterations of multiple dispatch: "
);
        long startTime = new Date().getTime();
        for (int iter = 0; iter < ITERATIONS; iter++ ) {
            for (int i = 0; i < names.length; i++) {
                for (int j = 0; j < names.length; j++) {
                    Real result = values[i].multiply1(values[j]);
                }
            }
        }
        long endTime = new Date().getTime();
        System.out.println((endTime - startTime) + " ms");
    }

    public static void testMultiply2() {
        System.out.print( ITERATIONS + " iterations of typecases: " );
        long startTime = new Date().getTime();
        for (int iter = 0; iter < ITERATIONS; iter++ ) {
            for (int i = 0; i < names.length; i++) {
                for (int j = 0; j < names.length; j++) {
                    Real result = values[i].multiply2(values[j]);
                }
```

```
            }
        }
        long endTime = new Date().getTime();
        System.out.println((endTime - startTime) + " ms");
    }

    public static void testMultiply3() {
        System.out.print( ITERATIONS + " iterations of double dispatch: " );
        long startTime = new Date().getTime();
        for (int iter = 0; iter < ITERATIONS; iter++ ) {
            for (int i = 0; i < names.length; i++) {
                for (int j = 0; j < names.length; j++) {
                    Real result = values[i].multiply3(values[j]);
                }
            }
        }
        long endTime = new Date().getTime();
        System.out.println((endTime - startTime) + " ms");
    }

    public static int ITERATIONS = 1000;
}
```

A.3. Raw Data for Tests

```
$ java evaluation.speed.TestOpenClassDispatch 100000
100000 iterations of nothing: 0 ms
100000 iterations of external method dispatchTest: N = 5, 270 ms
100000 iterations of external method dispatchTest: N = 7, 311 ms
100000 iterations of external method dispatchTest: N = 341, 15542 ms
100000 iterations of external method prettyPrint: N = 5, 1322 ms
100000 iterations of external method prettyPrint: N = 7, 1763 ms
100000 iterations of external method prettyPrint: N = 341, 129135 ms
100000 iterations of external method size: N = 5, 251 ms
100000 iterations of external method size: N = 7, 310 ms
100000 iterations of external method size: N = 341, 15783 ms
100000 iterations of visitor dispatch tester: N = 5, 50 ms
100000 iterations of visitor dispatch tester: N = 7, 80 ms
100000 iterations of visitor dispatch tester: N = 341, 3265 ms
100000 iterations of visitor pretty-printer: N = 5, 1792 ms
100000 iterations of visitor pretty-printer: N = 7, 2254 ms
100000 iterations of visitor pretty-printer: N = 341, 141804 ms
100000 iterations of visitor sizer: N = 5, 120 ms
100000 iterations of visitor sizer: N = 7, 160 ms
100000 iterations of visitor sizer: N = 341, 7461 ms
100000 iterations of regular method internalPrettyPrint: N = 5, 1061 ms
100000 iterations of non-extensible pretty-print visitor: N = 5, 1703 ms

$ java evaluation.speed2.TestMultipleDispatch 1000000
1000000 iterations of doing nothing: 140 ms
1000000 iterations of multiple dispatch: 4306 ms
1000000 iterations of typecases: 4307 ms
1000000 iterations of double dispatch: 2704 ms
```

REFERENCES

AGRAWAL, R., DEMICHIEL, L. G., AND LINDSAY, B. G. 1991. Static type checking of multi-methods. *ACM SIGPLAN Notices 26,* 11 (Nov.), 113–128. OOPSLA '91 Conference Proceedings, Andreas Paepcke (editor), October 1991, Phoenix, Arizona.

ARNOLD, K., GOSLING, J., AND HOLMES, D. 2000. *The Java Programming Language Third Edition*, Third ed. Addison-Wesley, Reading, MA.

BARENDREGT, H. P. 1984. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Publishing Co., New York, NY. Revised Edition.

BOURDONCLE, F. AND MERZ, S. 1997. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 302–315.

BOYLAND, J. AND CASTAGNA, G. 1997. Parasitic methods: Implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97, Atlanta*. ACM SIGPLAN Notices, vol. 32(10). ACM, New York, 66–76.

BRUCE, K., CARDELLI, L., CASTAGNA, G., GROUP, T. H. O., LEAVENS, G. T., AND PIERCE, B. 1995. On binary methods. *Theory and Practice of Object Systems 1,* 3, 221–242.

CARDELLI, L. 1988. A semantics of multiple inheritance. *Information and Computation 76,* 2/3 (February/March), 138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

CARDELLI, L. 1997. Program fragments, linking, and modularization. In *Conference Record of POPL 97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*. ACM, ACM, New York, NY, 266–277.

CASTAGNA, G. 1995. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst. 17,* 3, 431–447.

CASTAGNA, G. 1997. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston.

CASTAGNA, G., GHELLI, G., AND LONGO, G. 1995. A calculus for overloaded functions with subtyping. *Information and Computation 117,* 1 (Feb.), 115–135. A preliminary version appeared in *ACM Conference on LISP and Functional Programming*, June 1992 (pp. 182–192).

CHAMBERS, C. 1992. Object-oriented multi-methods in Cecil. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands,* O. L. Madsen, Ed. Lecture Notes in Computer Science, vol. 615. Springer-Verlag, New York, NY, 33–56.

CHAMBERS, C. 1995. The Cecil language specification and rationale: Version 2.0. Available from http://www.cs.washington.edu/research/projects/cecil/.

CHAMBERS, C. 1998. Towards Diesel, a next-generation OO language after Cecil. Invited talk, the *Fifth Workshop of Foundations of Object-Oriented Langauges,* San Diego, California.

CHAMBERS, C. AND CHEN, W. 1999. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*. ACM SIGPLAN Notices, vol. 34(10). ACM, Denver, CO, 238–255.

CHAMBERS, C. AND LEAVENS, G. T. 1995. Typechecking and modules for multi-methods. *TOPLAS 17,* 6 (Nov.), 805–843.

CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*. ACM SIGPLAN Notices, vol. 35(10). ACM, New York, 130–145.

COOK, W. R. 1991. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990,* J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 489. Springer-Verlag, New York, NY, 151–178.

DUTCHYN, C., LU, P., SZAFRON, D., BROMLING, S., AND HOLST, W. 2001. Multi-dispatch in the Java Virtual Machine: Design and implementation. In *Sixth Conference on Object-Oriented Technologies and Systems (COOTS), San Antonio, Texas.* USENIX, Berkeley, CA. Obtained from http://citeseer.nj.nec.com/dutchyn01multidispatch.html.

FEINBERG, N., KEENE, S. E., MATHEWS, R. O., AND WITHINGTON., P. T. 1997. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass.

FINDLER, R. B. AND FLATT, M. 1999. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM SIGPLAN Notices, vol. 34(1). ACM, New York, 94–104.

FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. 1992. *Essentials of Programming Languages*. McGraw-Hill Book Co., New York, NY.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass.

HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming (a critique of pure objects). In *OOPSLA 1993 Conference Proceedings,* A. Paepcke, Ed. ACM SIGPLAN Notices, vol. 28(10). ACM Press, New York, 411–428.

HÖLZLE, U. 1993. Integrating independently-developed components in object-oriented languages. In *Object-Oriented Programming 7th European Conference ECOOP '93 Kaiserslautern, Germany, Proceedings*. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, New York, NY, 36–56.

INGALLS, D. H. H. 1986. A simple technique for handling multiple polymorphism. *ACM SIGPLAN Notices 21,* 11 (Nov.), 347–349. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

KELSEY, R., CLINGER, W., AND (EDITORS), J. R. 1998. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33,* 9 (Sept.), 26–76.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of aspectj. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary,* J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Berlin Heidelberg, 327–353.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, M. Akcsit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, New York, NY, 220–242.

KRISHNAMURTHI, S., FELLEISEN, M., AND FRIEDMAN, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP'98—Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, New York, 91–113.

LEAVENS, G. T. AND MILLSTEIN, T. D. 1998. Multiple dispatch as dispatch on tuples. In *OOPSLA '98 Conference Proceedings*. ACM SIGPLAN Notices, vol. 33(10). ACM, New York, 374–387.

LINDHOLM, T. AND YELLIN, F. 2000. *The Java Virtual Machine Specification,* Second ed. Addison-Wesley Publishing Co., Reading, MA.

MARTIN, R. C. 1998. Acyclic visitor. In *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, Reading, Mass., 93–104.

MILLSTEIN, T. AND CHAMBERS, C. 1999. Modular statically typed multimethods. In *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal,* R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, New York, NY, 279–303.

MUGRIDGE, W. B., HOSKING, J. G., AND HAMER, J. 1991. Multi-methods in a statically-typed programming language. In *ECOOP '91 Conference Proceedings, Geneva, Switzerland,* P. America, Ed. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, New York.

NORDBERG, M. E. 1998. Default and extrinsic visitor. In *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, Reading, Mass., 105–123.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 146–159.

PAEPCKE, A. 1993. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, Cambridge, Mass.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec.), 1053–1058.

PARNAS, D. L. 1975. Software engineering or methods for the multi-person construction of multi-version programs. In *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, 25-27 September, 1974*, C. E. Hackl, Ed. Lecture Notes in Computer Science, vol. 23. Springer-Verlag, New York, NY, 225–235.

PARR, T. J. AND QUONG, R. W. 1994. Adding semantic and syntactic predicates to LL(k)–pred-LL(k). In *Proceedings of the 5th International Conference on Compiler Construction, Edinburgh, Scotland*, P. A. Fritzson, Ed. Lecture Notes in Computer Science, vol. 786. Springer-Verlag, Berlin, New York, 263–277.

REYNOLDS, J. C. 1975. User-defined types and procedural data structures as complementary approaches to type abstraction. In *New Directions in Algorithmic Languages*, S. A. Schuman, Ed. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 157–168.

RUBY, C. AND LEAVENS, G. T. 2000. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*. ACM SIGPLAN Notices, vol. 35(10). ACM, New York, 208–228.

SCHMIDT, D. A. 1994. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Mass.

SHALIT, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass.

STEELE JR., G. L. 1990. *Common LISP: The Language*, Second ed. Digital Press, Bedford, Mass.

STROUSTRUP, B. 1997. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass.

VLISSIDES, J. 1999. Visitor into frameworks. *C++ Report 11*, 10 (November/December), 40–46.

ZENGER, M. AND ODERSKY, M. 2001. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming (ICFP '01), Firenze, Italy.* ACM SIGPLAN Notices. ACM, New York, NY.