

Relaxed MultiJava: Balancing Extensibility and Modular Typechecking

Todd Millstein, Mark Reay, and Craig Chambers

Department of Computer Science and Engineering

University of Washington

Box 352350

Seattle, WA 98195-2350 USA

{todd,mreay,chambers}@cs.washington.edu

ABSTRACT

We present the rationale, design, and implementation of Relaxed MultiJava (RMJ), a backward-compatible extension of Java that allows programmers to add new methods to existing classes and to write multimethods. Previous languages supporting these forms of extensibility either restrict their usage to a limited set of programming idioms that can be modularly typechecked (and modularly compiled) or simply forego modular typechecking altogether. In contrast, RMJ supports the new language features in a virtually unrestricted form while still providing mostly-modular static typechecking and fully-modular compilation. In some cases, the RMJ compiler will warn that the potential for a type error exists, but it will still complete compilation. In that case, a custom class loader transparently performs load-time checking to verify that the potential error is never realized. RMJ's compiler and custom loader cooperate to keep load-time checking costs low. We report on qualitative and quantitative experience with our implementation of RMJ.

1. INTRODUCTION

The design of a programming language must balance several competing goals. One important goal is the ability to organize software into separate modules, each of which can be reasoned about (e.g. typechecked, or compiled) separately from the implementations of other modules. This kind of modular checking allows software components to be developed and checked for correctness once and then reliably reused in many future contexts.

Another important goal is the ability to easily extend existing software with new capabilities, without requiring the existing software to be modified. Standard object-oriented languages gain great expressive power by allowing a class to be defined as an extension (i.e., a subclass) of an existing class, without modifying the existing class or any of its clients. More advanced object-oriented languages, including Common Lisp [Steele Jr. 90, Paepcke 93], Dylan [Shalit 96], Cecil [Chambers 92, Chambers 93], AspectJ [Kiczales et al. 97, Kiczales et al. 01], and Hyper/J [Harrison & Ossher 93, Ossher & Tarr 00], support several additional forms of extensibility, such as adding new methods to existing classes, adding statements before or after existing methods, adding new superclasses to existing classes, and/or allowing methods to dynamically dispatch on the run-time classes of their arguments (which is a kind of extension to those argument classes).

Unfortunately, modular reasoning is in conflict with flexible extensibility. In general, the more a module can be extended from the outside, the fewer properties can be proven about the module separately from those extensions. For example, traditional statically typed object-oriented languages check that each operation is properly implemented on a class-by-class basis. This checking

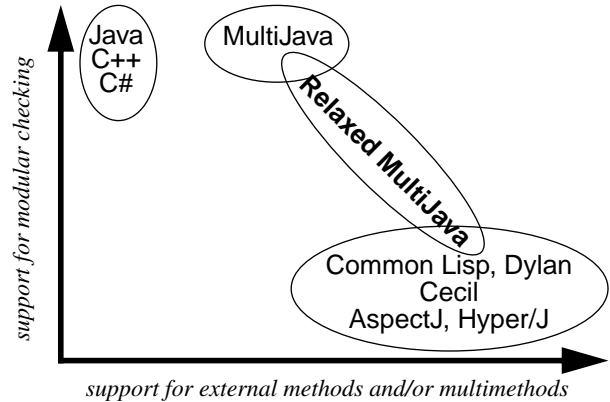


Figure 1: Tradeoffs between extensibility and modular checking

ensures that dispatch errors such as “message not understood” and “message ambiguous” can never occur on message sends at run time. But if new methods can be added to existing classes in an unrestricted manner, then it is easy to introduce message dispatch errors that elude modular detection [Millstein & Chambers 99].

Because of these conflicts, each language design represents a particular tradeoff between the amount of extensibility allowed and the amount of modular typechecking supported. Most existing languages have been biased toward one or the other extreme. For example, standard object-oriented languages support modular class-by-class typechecking but only support subclassing-based extensibility. At the opposite end of the spectrum, the advanced languages listed earlier support several additional kinds of extensibility. However, the cost of this greater extensibility has been a loss of modular static checking and compilation; these kinds of languages require whole-program information to perform typechecking (if they support static typechecking at all) and perhaps to perform compilation as well. Figure 1 *very roughly* sketches these extremes.

In previous work with other colleagues, we developed MultiJava [Clifton et al. 00, Clifton 01, Mul], an extension to Java that augments Java's subclassing-based extensibility with the ability to add methods (called *external methods*) to existing classes (called *open classes* [Chambers 98]) and the ability to write methods (called *multimethods*) that can dispatch on argument classes in addition to the receiver class. MultiJava supports these additional features while retaining Java's modular typechecking and compilation schemes. To do so, MultiJava restricts the ways in which the new language features may be used to a particular set of extensibility idioms that are compatible with modular checking.

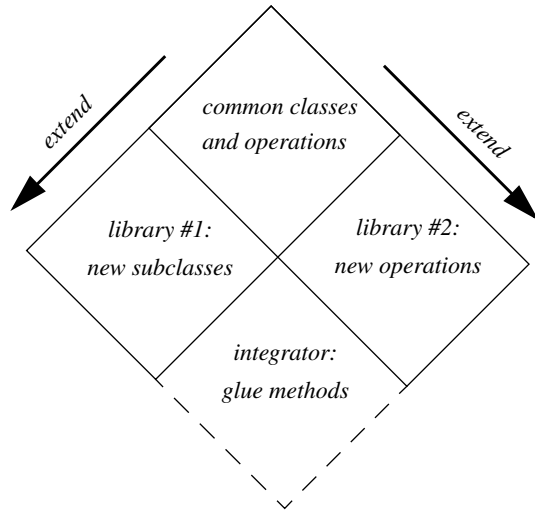


Figure 2: Completing the extensibility diamond

As a consequence of MultiJava’s insistence on fully modular typechecking, there are several useful forms of extensibility that are simply disallowed. For example, MultiJava does not allow an external method to be *abstract*, even when the method is added to an existing abstract class. This is because, given a strictly modular view of the program, in general it is not possible to guarantee that the new operation is implemented by all concrete subclasses of the abstract class. Because there is the *potential* for the new operation to be incomplete given only partial program information, MultiJava conservatively rejects the abstract external method declaration. However, it is quite possible that the programmer has ensured the new operation is fully implemented. For example, the programmer may wish to organize code into separate files for different operations, as opposed to the traditional organization by class. Even if the programmer properly defines each file’s operation for all relevant concrete classes in the program, MultiJava will still reject this organization.

MultiJava also requires all external method declarations that belong to the same operation to be written in a single file. This is because, given a strictly modular view, it would not otherwise be possible to guarantee the absence of duplicate or ambiguous external method declarations for that operation. Again, because there is the *potential* for an ambiguity given only partial program information, MultiJava conservatively rejects “free-standing” external method declarations. However, it is also possible that free-standing external methods are completely safe, and in practice there are programming situations that need them. For example, a client of two independently developed libraries may need to provide implementations of operations defined in one library for concrete classes defined in the other library; in other words, the client needs to “complete the diamond” set up by the two independent extensions, as illustrated in Figure 2. We sometimes refer to free-standing methods as “glue methods,” since they serve to combine two separate libraries. Even if the programmer ensures that glue methods do not cause ambiguities, MultiJava will still reject this programming idiom.

As a final example, MultiJava disallows writing methods that dispatch (either as an argument of a multimethod or as the receiver of an external method) on an interface, because MultiJava’s modular view of a program is not sufficient to guarantee the

absence of ambiguities caused by multiple inheritance. However, again we have only the *potential* for an ambiguity; it is quite possible for the programmer to have arranged the code so that no multiple-inheritance ambiguities can arise. Unfortunately, it is common practice for a third-party library to expose only interfaces to clients, rather than the classes implementing those interfaces. MultiJava’s restriction prevents clients from augmenting such a library with new external methods or multimethods.

In this paper we present the design and implementation of Relaxed MultiJava (RMJ). Like MultiJava, RMJ augments Java with external methods and multimethods, and it provides modular typechecking and compilation. At the same time, RMJ supports nearly arbitrary usage of the new features, including expression of the three examples described above that MultiJava cannot allow. These properties are achieved by giving programmers explicit control over the tradeoff between extensibility and modular reasoning, rather than having the language legislate one or the other extreme.

The key technical principle underlying RMJ’s design is to treat the modular detection of the *potential* for a message dispatch error as producing merely a compile-time *warning*. For any operation flagged at modular compile time as potentially incompletely or ambiguously implemented, the programmer can choose to resolve the problem and acquire a guarantee of modular type safety. Alternatively, the programmer can retain the extra expressiveness that triggered the warning. In that case, the operation will undergo more checking at load time, to ensure that the operation is in fact properly implemented. We employ a custom class loader to perform this load-time checking. RMJ’s strategy allows the expression of many more idioms than are expressible in MultiJava, but it still ensures that (a) all message dispatch errors are detected no later than load time, and (b) the programmer is always aware at modular compile time of the potential for any load-time errors. MultiJava’s type system falls out as a special case of RMJ, corresponding to a scenario in which all compile-time message dispatch warnings are treated as errors by the programmer.

RMJ has the following novel collection of characteristics:

- RMJ is strictly more expressive than MultiJava, which in turn is strictly more expressive than Java. Aside from a few compilation challenges discussed later, RMJ allows arbitrary usage of external methods and multimethods.
- RMJ provides the same modular static assurances as MultiJava, because RMJ modularly and statically identifies and reports to the programmer the same problems as MultiJava. If MultiJava would report no errors to the programmer, then RMJ will report no errors to the programmer, and no errors can occur, even at load time. But where MultiJava would reject a program, RMJ might instead warn of a potential problem, allowing the programmer to take responsibility for avoiding it.
- For all compile-time warnings, the RMJ class loader will check at class load time whether the potential error actually occurs for the program being linked. This check can be viewed as a natural augmentation of the normal class verification check in the standard Java class loader. If a class or external method loads successfully, then there can be no message dispatching errors involving that class or method. Such load-time checking is qualitatively better than run-time checking of each message send, even when (as in Java’s case) class loading can occur at run time. Run-time checking can never prove that some future message send won’t fail, whereas load-time checking guarantees that, for those classes that are loaded, there cannot be any message send, on any future execution path, that fails.

- The combination of compile-time warnings and load-time checks provides programmers with fine-grained control over the tradeoff between expressiveness and modular reasoning. If a static warning is signaled, the programmer can choose to resolve the problem modularly, as (s)he would be forced to do in MultiJava, thereby gaining a modular guarantee of type safety. Alternatively, the programmer can leave the code as originally written, gaining expressiveness at the cost of additional responsibility to avoid a load-time error.
- RMJ's load-time checking typically occurs incrementally as a program runs, because of Java's lazy class loading style. RMJ also includes a "preloader" tool that statically checks an application for load-time errors, prior to running it.
- As with Java and MultiJava, RMJ source code is compiled into standard Java class files modularly, one file at a time. Therefore, RMJ source and compiled files interoperate seamlessly with Java source and compiled files.
- RMJ's compiler and class loader collaborate to make the necessary load-time checking efficient, incremental, and mostly a "pay-as-you-go" proposition.

An implementation of RMJ is freely available for download and experimentation [Mul].

The next section presents the design of the RMJ language. Section 3 describes our implementation strategy, including compiler support and the structure of the RMJ class loader. Section 4 assesses our work, presenting the results of some qualitative experience using the language and quantitative performance experiments. Section 5 describes previous work on increasing the modular extensibility of traditional object-oriented languages. Section 6 concludes with a discussion of future work.

2. LANGUAGE DESIGN

This section informally describes the RMJ language. Its syntax extends that of MultiJava for expressing external methods and multimethods in Java. Both RMJ and MultiJava are explicitly designed to be as small extensions to Java as possible, to make it easier for programmers to learn and adopt the new features. However, these syntactically small extensions offer significant new abilities to organize and extend programs. (More complete descriptions of the MultiJava language can be found elsewhere [Clifton et al. 00, Clifton 01].)

Throughout this section we will use a running example, inspired by an example due to Krishnamurthi [Krishnamurthi et al. 98]. Imagine that one author develops an abstract Shape class, and two independent developers each provide concrete implementations for Rectangle and Circle, as shown in Figure 3. The draw method relies on the abstract OutputDevice class in the OutputPackage package (not shown).

2.1 Preliminaries

It is useful to consider the methods in an RMJ (or Java) program to be implicitly partitioned into a set of *operations* (sometimes referred to as *generic functions* [Moon 86, Bobrow et al. 86, Paepcke 93]). Each operation is a collection of methods that have the same name and type signature. A method *m* that does not override any other method introduces a new operation, and all methods that override *m* belong to its operation. For example, the draw method in Shape of Figure 3 introduces an operation, and the other two draw methods in the figure also belong to it.

```
package ShapePackage;
import OutputPackage.*;
public abstract class Shape {
    ... generic operations on shapes ...
    public abstract void draw(OutputDevice d);
}

-----

package RectanglePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Rectangle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputDevice d) {
        ... code for drawing a Rectangle ...
    }
}

-----

package CirclePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Circle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputDevice d) {
        ... code for drawing a Circle ...
    }
}
```

Figure 3: Shape and two implementations

Each syntactic call site *s* in a program invokes a single operation's methods. The mapping from *s* to its associated operation *o* is determined statically, based on the static types of the receiver and other arguments to the call. When a message send occurs at *s* dynamically, the *most-specific applicable method* belonging to *o* is chosen. In the absence of multimethod dispatch, which is discussed in Section 2.3, a method is applicable to the message send if the class of the actual receiver is either the method's receiver class or some subclass. The most-specific applicable method is the unique applicable method that overrides all other applicable methods.

Two kinds of message dispatch errors are possible dynamically. If a message send has no applicable methods, then a "message not understood" error occurs. If a message send has applicable methods but no most-specific one, then a "message ambiguous" error occurs. Java's static typechecking guarantees that these errors can never occur by ensuring that each operation is properly implemented: it has a most-specific applicable method for every possible type-correct concrete receiver. For example, a static error would be signaled if Rectangle's draw method in Figure 3 were removed.

2.2 External Methods

RMJ and MultiJava allow new methods to be added to existing classes from the outside. For example, if a client of the Shape library wishes to view Shapes as providing an area operation, the client can program such an extended view of Shapes by writing a new file containing one or more *external method declarations* for area, as shown in Figure 4. In this example, the client knows about the Rectangle and Circle subclasses of

```

package AreaPackage;

import ShapePackage.*;
import RectanglePackage.*;
import CirclePackage.*;

public double Shape.area() {
    ... default implementation ...
}

public double Rectangle.area() {
    return width() * height();
}

public double Circle.area() {
    return Math.PI * radius() * radius();
}

```

Figure 4: External area methods

```

package TrianglePackage;

import ShapePackage.*;
import OutputPackage.*;
import AreaPackage.area;

public class Triangle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputDevice d) {
        ... code for drawing a Triangle ...
    }

    public double area() {
        return base() * height() / 2;
    }
}

```

Figure 5: Subclass area method

Shape and provides appropriate area methods for them, along with a default method that handles any other Shape subclasses that might exist.

We call the *area* operation *external* because its introducing method is external. In RMJ and MultiJava, external operations are scoped. To use the new *area* operation, client code must import it, just as classes are imported. The last import declaration in Figure 5 provides Triangle access to the *area* operation. Once imported, *area* is treated just like any other operation on Shapes. It can be invoked using Java's normal message-send syntax; there is no distinction to clients between the "original" operations of Shape (like *draw*) and the externally added ones (like *area*). Methods of external operations can also be overridden in subclasses, like other methods. For example, in Figure 5 the Triangle subclass of Shape includes an overriding implementation of *area* as a regular method inside its class declaration.

The ability to add methods to existing classes is a powerful and recurring idiom. The visitor design pattern [Gamma et al. 95] was developed in part to overcome the inability of existing mainstream languages to add new "visiting" operations to existing classes. The separation of code into multiple, orthogonal concerns, as in role-based programming [Andersen & Reenskaug 92, VanHilst & Notkin 96, Smaragdakis & Batory 98], subject-oriented programming [Harrison & Ossher 93, Ossher & Tarr 00], and aspect-oriented programming [Kiczales et al. 97, Kiczales et al. 01], is also dependent on the ability to organize methods not by class but by concern, and then to add these methods to the underlying classes from the outside. Even when it would be possible to put all methods into their class, such as when

developing an application from scratch, it may still be desirable to modularize some of the source code by operation.

A key strength of MultiJava is that each of the files in Figures 3-5 can be typechecked modularly, given only the interfaces of the *visible* classes and external operations, which are those that are referenced by a given file. For example, the *area* methods in Figure 4 are typechecked in the context of the interfaces for the classes in Figure 3, but without access to Triangle (which may not even have been written yet). If typechecking passes on each file, then every operation in the program is guaranteed to be properly implemented, so run-time message dispatch errors will not occur. In order to make this strong guarantee, MultiJava imposes significant limitations on the kinds of external methods that can be written. In contrast, RMJ provides the same modular checking as MultiJava but does not impose the associated limitations, instead transparently providing additional load-time safety checks as necessary. The following subsections describe three extensions that RMJ makes to MultiJava's external methods.

2.2.1 Abstract External Methods

It is natural to allow external methods of abstract classes to be abstract. For example, it may be desirable to declare Shape's *area* method abstract; Figure 6 illustrates how this is programmed in RMJ. Abstract external methods free the programmer from having to provide a default *area* implementation, for which there may be no reasonable semantics. They also allow the programmer to document the requirement that all concrete subclasses provide an appropriate *area* implementation.

However, it is difficult to preserve fully modular typechecking in the face of abstract external methods. For example, suppose the Triangle class of Figure 5 did not import *area* nor include an overriding *area* method. Then neither the Triangle class nor the *area* operation are visible to one another modularly. If the version of the *area* operation in Figure 6 is used, we can get a "message not understood" error at run time, if *area* is ever invoked on a Triangle.

MultiJava addresses this problem by simply disallowing abstract external methods, thereby ensuring that each external operation has a default method implementation. Unfortunately, a reasonable default implementation of an operation does not always exist. Unless the set of operations available on Shape is very rich, it is unlikely that any useful *area* default implementation can be written. Therefore the default implementation's body will probably be forced to simply throw an exception. Such a default implementation satisfies MultiJava's modular typechecker, but only by creating the potential for a run-time error which is not much different than the "message not understood" error that the default implementation is written to prevent! (MultiJava's approach works well for operations where overriding methods merely provide more efficient or customized implementations of a default algorithm, such as the union of two sets, but not for operations where the overriding methods define the appropriate behavior of the operation for the subclass.)

In contrast, RMJ allows abstract external methods to be written, signaling only compile-time *warnings* rather than compile-time *errors*. When the file containing the *area* methods in Figure 6 is compiled, the programmer will be issued a warning about the *potential* for *area* to be incompletely implemented, but the file will be compiled successfully. As long as all concrete subclasses of Shape loaded into the program define or inherit an implementation of *area*, the program will be correct and the potential for an incomplete implementation will not have been realized. However, if

```

package AreaPackage;

import ShapePackage.*;
import RectanglePackage.*;
import CirclePackage.*;

public abstract double Shape.area();

public double Rectangle.area() {
    return width() * height();
}

public double Circle.area() {
    return Math.PI * radius() * radius();
}

```

Figure 6: Abstract external methods in RMJ

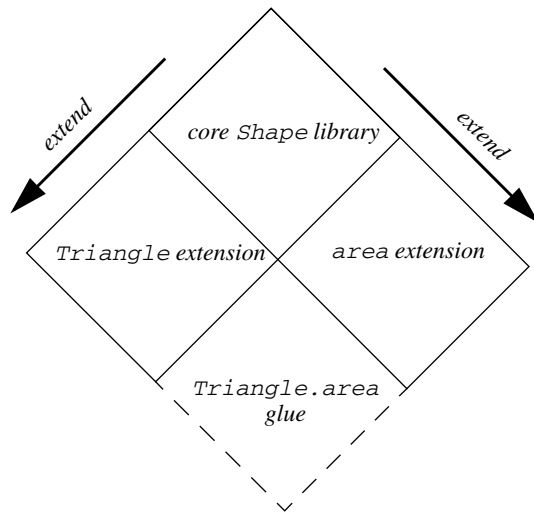


Figure 7: Completing the extensibility diamond

a concrete subclass of Shape is loaded that does not override the abstract area method declaration, then a load-time verification error will be reported. RMJ's combination of compile-time and load-time checking is sufficient to ensure that all operations are properly implemented. Therefore, a program that passes RMJ's compile-time and load-time checks will never generate any dispatching errors when messages are sent.

2.2.2 Glue Methods

Suppose again that the area and triangle libraries are two independent augmentations to the original shape hierarchy, so that the Triangle class wouldn't know about the area operation and wouldn't have an area method inside it. This scenario is sketched in Figure 7. As described above, if both the area operation in Figure 6 and the revised Triangle class are loaded into the same program, a load-time error will be triggered. To resolve this problem, the integrator of the two independently developed libraries must be able to provide additional "glue" code that makes the libraries work together, i.e., that "completes the diamond." For example, Figure 8 shows a new file that defines the external method enabling the area operation of Figure 6 to interoperate with Triangle, without modifying either library (or retypechecking either library or even having source access to either library).¹

1. Even if area had a default implementation, as in Figure 4, glue methods would still be useful, allowing clients to customize the integration of the area and triangle libraries.

```

package TriangleAndAreaGluePackage;

import TrianglePackage.*;
import AreaPackage.area;

public double Triangle.area() {
    return base() * height() / 2;
}

```

Figure 8: Glue external methods in RMJ

We refer to the area method in Figure 8 as a *glue method*, because it glues together an existing class with an existing operation. More precisely, a glue method is an external method that does not reside in the same file that introduces the method's associated operation. For example, the area method in Figure 8 belongs to the operation that was introduced in Figure 6.

Unfortunately, it is difficult with a purely modular view to ensure that there are not any duplications or ambiguities between the glue method in Figure 8 and the other methods in the area operation. For example, although the glue method in Figure 8 is not ambiguous with the area methods in Figure 6, if an unseen file contains another area glue method for Triangle, at run time a "message ambiguous" error will occur when area is invoked on a Triangle instance. Because of these kinds of problems, MultiJava does not allow glue methods to be written. It instead requires all the external methods for a particular operation to be written in the file that introduces the operation, allowing an operation's external methods to be typechecked as a unit, thereby preserving modular typechecking.

In contrast, RMJ allows glue methods to be written but issues a compile-time warning that there is the *potential* for duplicate or ambiguous methods to appear in other files. RMJ will still compile the glue methods successfully. The class loader will then verify as glue methods are loaded that there are no duplicates or ambiguities.

An unusual issue in the design of glue methods is the need to determine how they interact with Java's lazy loading capabilities. A class is typically loaded in Java implementations upon first reference (e.g. when an instance is created). Similarly, in RMJ and MultiJava, an external operation is loaded in a program simply by referencing it by name (e.g. in a message send to that operation). Referencing an external operation has the effect of loading the operation's introducing method, as well as all overriding methods in the same file. However, by its nature a glue method is written separately from its operation, so it will not be loaded by this scheme. Furthermore, individual methods are never named directly in programs -- a method is always invoked indirectly via message sends to its associated operation.

To address this problem, our custom class loader accepts a list of all the files containing glue methods to be included in a given program. Before loading the program's first class (the one containing the main method), the class loader records the existence of each glue method, but it does not load any glue methods. Each glue method will be loaded as soon as it is *reachable*, meaning that the method's operation, receiver, and argument types have all been loaded. This strategy ensures that a glue method is not loaded before it is capable of being invoked, in keeping with Java's lazy loading scheme. At the same time, the strategy still maintains a kind of monotonicity in the meaning of operations: the method chosen by invoking an operation with a given receiver and arguments cannot not change during the course of a program, even if new methods are added to the operation through later class loading. Implementation details of our strategy for loading glue methods are provided in Section 3.

While RMJ supports glue methods belonging to external operations like `area`, it currently does not support glue methods belonging to regular “internal” operations like `draw`. Glue methods for external operations allow us to integrate separately developed class hierarchies and external operations, which was our goal. However, supporting glue methods on internal operations would enable additional kinds of useful expressiveness, particularly in the presence of multimethods (described in Section 2.3). Unfortunately, it is challenging to modularly compile glue methods belonging to internal operations in a way that is efficient and that interoperates seamlessly with existing Java source and compiled files. We leave this to future work.

2.2.3 External Methods on Interfaces

Since external methods are declared and compiled outside of their receivers, it is reasonable to allow programmers to write external methods on interfaces in addition to classes. RMJ accordingly allows this idiom to be expressed. However, external methods on interfaces pose a challenge for modular typechecking, because interfaces support multiple inheritance. Therefore, two external methods on interfaces may appear to be unambiguous but may cause a run-time “message ambiguous” error if a later class implements both interfaces [Millstein & Chambers 99].

MultiJava handles this modularity problem by disallowing overriding external methods from being added to interfaces. MultiJava allows a new operation to be introduced on an interface and given a default implementation, but it does not allow overriding methods to be added to “subinterfaces.” For example, if `Shape`, `Rectangle`, and `Circle` were defined as interfaces rather than classes in Figure 3, MultiJava would still allow the `area` operation of Figure 4 to be introduced on the `Shape` interface, but it would not allow overriding methods to be defined on `Rectangle` or `Circle`. Unfortunately, this restriction prevents clients from usefully augmenting libraries that export only interfaces, rather than the underlying implementation classes.

In contrast, RMJ allows arbitrary declaration of external methods on interfaces, but it produces a compile-time warning for overriding external methods on an interface. In the scenario where `Shape`, `Rectangle`, and `Circle` were defined as interfaces rather than classes, when compiling the collection of `area` methods from Figure 4, RMJ would warn that the `Rectangle` and `Circle` methods might be ambiguous for some unseen class.² The programmer could then decide to program the `area` operation in a different way, to avoid the potential for an ambiguity, or the programmer could decide that, for this application domain, there is no potential ambiguity (i.e., that there won’t be any shapes that are both `Rectangles` and `Circles`); the RMJ class loader will confirm the programmer’s understanding in this case. RMJ gives the programmer the flexibility to tradeoff modular guarantees against increased expressiveness on an operation-by-operation basis.

2.3 Multimethods

RMJ and MultiJava also extend Java by allowing message dispatch to depend upon the run-time classes of the arguments of the message in addition to the receiver; this is called *multiple dispatching* (as opposed to the *single dispatching* of traditional receiver-based method lookup). To exploit multiple dispatching, a

2. If a multiply inheriting class were visible when compiling the file containing the `area` methods, the compiler would generate a compile-time error, not just a warning.

```
package RectanglePackage;
import ShapePackage.*;
import OutputPackage.*;

public class Rectangle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputStream d) {
        ... code for drawing a Rectangle ...
    }

    public void draw(OutputStream@BWPrinter p)
    { ... code for drawing a Rect. on a b&w printer ...
    }
}

-----

package CirclePackage;
import ShapePackage.*;
import OutputPackage.*;

public class Circle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputStream d) {
        ... code for drawing a Circle ...
    }

    public void draw(OutputStream@BWPrinter p)
    { ... code for drawing a Circle on a b&w printer ...
    }
}
```

Figure 9: draw multimethods

(possibly external) method can add a *specializer* to one or more of its arguments, which restricts the method to only apply to message sends whose arguments are instances of the specializing classes (or their subclasses); methods with argument specializers are called *multimethods*.

For example, consider the `draw` operation for Shapes in Figure 3. It may be useful to have special drawing functionality for particular kinds of output devices. Figure 9 shows revised `Rectangle` and `Circle` classes, each with a new multimethod for drawing on black-and-white printers. To specify a specializer, a formal argument is declared using the syntax `StaticType@SpecializerClass FormalName`. In the `Rectangle` class, the second `draw` method is *applicable* only if the dynamic class of the receiver is `Rectangle` (or a subclass) *and* the dynamic class of the argument is `BWPrinter` (or a subclass). Multimethods are completely orthogonal to Java’s static overloading mechanism, which uses the *static* types of the arguments at a call site in determining which *operation* the site invokes.

At run time, the most-specific applicable method is invoked, as described in Section 2.1. In the presence of multimethods, a method *m* overrides a method *n* if *m* and *n* differ in either their receiver or an argument specializer, *m*’s receiver is either *n*’s receiver or a subtype, and for each argument position *i*, *m*’s *i*th specializer is either *n*’s *i*th specializer or a subtype. In our example, if `draw` is sent to a `Rectangle` and a `BWPrinter`, then both `Rectangle` `draw` methods are applicable and the second one is chosen, since the methods have the same receiver but the second’s argument specializer is more specific (an unspecialized argument is equivalent to one specialized to the static type). Sending the `draw` message to a `Rectangle` and a `ColorPrinter`, however, will

invoke the first `Rectangle` draw method, since the second one is not applicable.

Multimethods are very useful for writing code that depends on the particular classes of more than just the receiver. In addition to operations like `draw`, *binary* operations like equality, addition, and set union, which accept two arguments of the same type, are good examples. Multimethods allow the arguments of a binary operation to be treated symmetrically and allow algorithm selection to be sensitive to the representations of both arguments. We have also found multimethods to be quite useful in event-based systems, where components register themselves to be notified when an event occurs. Notification consists of the invocation of a component's handle operation, passing the event as an argument. To define how events are dispatched, a component defines some number of handle multimethods, each of which specializes its event argument to the particular subclass of event to be handled.

As with external methods, MultiJava is able to modularly typecheck and compile files containing multimethods. If typechecks succeed on all files, then MultiJava guarantees that each operation is properly implemented. In the context of multimethod dispatch, this means that the operation has a most-specific applicable method for every possible type-correct tuple consisting of a concrete receiver and concrete arguments. As with external methods, however, MultiJava imposes some restrictions on how multimethods are written to ensure this ability to check multimethods modularly. Each concrete class is required to define or inherit a *singly-dispatched* implementation of each operation that it supports. For example, in the `Rectangle` class in Figure 9, the first draw method, which doesn't specialize on its argument, is required. If it were omitted, MultiJava would issue a compile-time error, because `draw` could be incompletely implemented if there exist output devices other than `BWPrinter`, for example `ColorPrinter`. With its strictly modular view, the MultiJava typechecker does not know whether such classes exist or not, so it conservatively rejects programs lacking these singly-dispatched default implementations. Unfortunately, as with the earlier `area` operation, it may be difficult to write a default implementation of the `draw` operation that does not simply throw an exception.

RMJ treats the absence of singly-dispatched default methods in a concrete subclass as a compile-time *warning* rather than a compile-time *error*. In our example, the default draw methods in `Rectangle` and `Circle` can be omitted, leaving only the draw multimethods, as in Figure 10. When each of the `Rectangle` and `Circle` files is compiled, as long as `draw` is implemented for all visible concrete subclasses of `OutputDevice`, the RMJ compiler will issue only a warning that there is the *potential* for `draw` to be incompletely implemented, but the file will be compiled successfully. As long as `BWPrinter` and its subclasses are the only concrete kinds of output devices loaded into the program, the program will be correct and the potential for an incomplete implementation will not be realized. However, if a different concrete subclass of `OutputDevice` is loaded, then a load-time verification error will be reported. As before, RMJ's combination of compile-time and load-time checking is sufficient to ensure that all operations are properly implemented. Therefore, a program that passes RMJ's compile-time and load-time checks will never generate any dispatching errors when messages are sent.

MultiJava also requires that a multimethod's argument specializer be a class, not an interface. This restriction ensures that there are no multiple-inheritance ambiguities that elude modular detection, just as did the restriction that overriding external methods be on classes, not interfaces. If the different kinds of printers were exported from

```
package RectanglePackage;
import ShapePackage.*;
import OutputPackage.*;

public class Rectangle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputDevice@BWPrinter p)
    { ... code for drawing a Rect. on a b&w printer ...
    }
}

-----
package CirclePackage;
import ShapePackage.*;
import OutputPackage.*;

public class Circle extends Shape {
    ... implementations of Shape methods ...
    public void draw(OutputDevice@BWPrinter p)
    { ... code for drawing a Circle on a b&w printer ...
    }
}
```

Figure 10: draw multimethods in RMJ

`OutputPackage` as interfaces rather than classes, then it would not be possible to write draw multimethods specializing on `BWPrinter`.

In contrast, RMJ allows arguments to specialize on interfaces as well as classes. Interface specialization leads to compile-time warnings about potential ambiguities (or compile-time errors about any visible ambiguities), backed up by load-time checking to verify that the potential ambiguities never occur in practice.

2.4 Discussion

In RMJ we have identified those restrictions of MultiJava that reflect only the potential for a message dispatch error and replaced them with warnings. RMJ still reports all real and potential errors as each file is modularly compiled. This contrasts with other systems that have comparable support for external methods or multimethods, which require whole-program information in order to typecheck and/or compile a file. For example, a class in AspectJ may be typechecked and compiled only when given all of the aspects that add external methods (among other things) to the class. RMJ's modular checking is particularly important when compiling library files whose clients are not yet known.

RMJ greatly expands the practical utility of external methods and multimethods as compared with MultiJava. External methods and multimethods can be written and organized in any grouping desired. Those organizations that satisfy MultiJava's modular typechecking restrictions are proven safe entirely modularly. The remainder must be confirmed with a load-time check, but in many cases, the only feasible alternatives to load-time checking are methods that simply throw run-time exceptions. The specialized class loader that performs these load-time checks in the context of Java's incremental class loading strategy is described in the next section.

RMJ's type system can be viewed as a variant of the *soft typing* approach [Cartwright & Fagan 91], but with load-time checks instead of run-time checks. Soft typing systems attempt to perform as much static checking as possible on programs written in a dynamically typed language like Scheme. Our work takes the

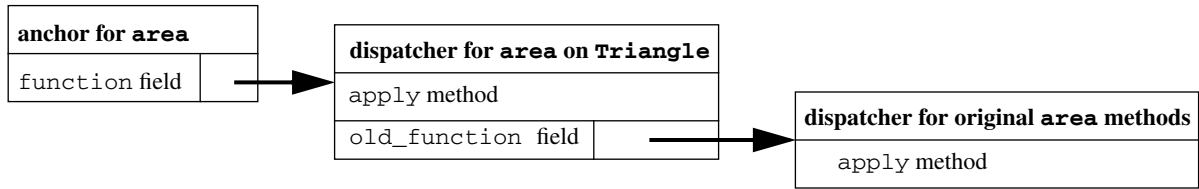


Figure 11: Structure of the implementation of *area* from Figures 4 and 5

opposite perspective: we relax a language that supports modular static typechecking to allow more expressiveness, inserting as few load-time checks as are necessary to ensure safety.

RMJ can serve as a general platform for experimenting with modular type systems for languages with external methods and multimethods. New modular type systems can be evaluated without changing the underlying expressiveness of the language. Rather, what changes is simply the factoring of the checks between compile time and load time. This approach would also be useful for other kinds of expressive languages that pose a challenge for modular typechecking, including aspect-oriented languages.

3. COMPILATION AND CLASS LOADING

RMJ source code compiles into regular Java bytecode classes, which are loaded by a custom class loader running on a standard Java virtual machine. This section explains how bytecode for RMJ is generated, loaded, and verified. We begin by briefly reviewing how MultiJava’s language extensions are compiled, then explain the additional compilation techniques used for RMJ, and finally describe RMJ’s custom class loader. A key feature of our compilation strategy is that the custom class loader only performs extra checking on an operation if the compiler was forced to emit a warning message about potential incompleteness or ambiguity problems for that operation. If the compiler can verify the correctness of all classes modularly, then the custom class loader will perform no load-time checking.

3.1 Compile-Time Extensions in MultiJava

RMJ is able to reuse most of the compilation techniques of MultiJava. To implement multiple dispatching, MultiJava merges all the methods that have the same receiver class but different argument class specializers into a single singly-dispatched bytecode method, with a series of `instanceof` tests selecting the right branch containing the body of one of the original multimethods. Clients continue to invoke operations containing multimethods in exactly the same way as before, thereby shielding clients from whether or not some operation has multimethods. This design also allows a MultiJava class to extend a regular Java class and override a regular Java method with a MultiJava multimethod, all transparently to the Java class and its existing Java clients.

To implement external operations, MultiJava generates an *anchor class* representing the operation. The external methods declared in the file that introduces the external operation are merged into a single bytecode method named `apply`, in which the original receiver has been converted into an additional argument. This `apply` method selects the right branch using a series of `instanceof` tests of the original receiver plus any other arguments with `specializer` classes.

As shown in Figure 5, subclasses of the receiver of an external operation can import the operation and then add additional methods to it. To allow an external operation to be extended by later classes in this way, the `apply` bytecode method for the group of methods

on an external operation from a single source file is put in a *dispatcher class*; the external operation’s anchor class then maintains a linked list of dispatcher class instances, in order of most specific to least specific. If a subclass adds one or more new methods to an existing external operation, the subclass methods are compiled into an `apply` method in their own dispatcher class, which is added to the front of the anchor class’s dispatcher list as part of the subclass’s static initialization code.

For example, consider the *area* external operation defined in Figure 4. The operation gets its own anchor class, and the three methods are merged into an `apply` method in a new dispatcher class. When *Triangle* of Figure 5 is compiled, a dispatcher class for its *area* method is created as well. *Triangle*’s static initialization code eventually adds this dispatcher to the front of the anchor class’s dispatcher list, resulting in the structure illustrated in Figure 11. When *area* is invoked, the head of the anchor class’s list of dispatchers is fetched, and its `apply` method is invoked. If none of the methods of the head dispatcher applies, then the `apply` method fetches the next dispatcher in the chain, and invokes its `apply` method recursively. Eventually, an applicable method will be found, because modular static typechecking has verified that the operation is completely implemented.³

More details on the implementation techniques of MultiJava are available in earlier papers [Clifton et al. 00, Clifton 01].

3.2 Compile-Time Extensions in RMJ

The RMJ compiler has two code-generation tasks beyond what the MultiJava compiler does. First, the RMJ compiler must generate appropriate bytecode for the additional features not supported by MultiJava. Second, the RMJ compiler must record information in the resulting class files to tell the RMJ class loader what checks to perform when each class is loaded. This information is conveyed through the extensible annotation mechanism already supported by Java’s class file format [Lindholm & Yellin 97]. That mechanism allows arbitrary strings to be included in compiled class files, indexed by user-defined keys.

3.2.1 Compiling RMJ Extensions

RMJ allows the method introducing an external operation to be abstract, as illustrated in Figure 6. Bytecode generation for abstract methods is simple: the abstract method is treated as if it has a body that simply throws a `RuntimeException`. A similar technique can be used to generate code when a concrete class lacks a singly dispatched method for some operation, as illustrated in Figure 10. Bytecode for the implicit abstract method can be generated, with a body that throws a `RuntimeException`.

3. This “chain of responsibility” [Gamma et al. 95] style works correctly and can be generated completely modularly and statically, but it is not as efficient as regular method invocation in Java. An alternative strategy worth investigating would generate a more efficient custom dispatcher method at load time, based on the current set of loaded dispatchers.

RMJ allows the receiver or an argument specialist of a method to be an interface. Bytecode generation is unaffected by this relaxation of MultiJava, since the existing strategy of testing for a method's applicability using `instanceof` tests works for interfaces as well as classes.

Finally, RMJ allows a method to be declared in a separate file from both its receiver class and its (external) operation. We compile each such glue method into an `apply` method in its own dispatcher class; this enables each glue method to be loaded separately, as it becomes reachable. A source file containing several glue methods is itself compiled into a *glue anchor class* akin to an external operation's anchor class. The glue anchor class is used to provide annotations to the class loader about the new glue methods, as described below.

In RMJ, it is possible for glue methods to override some existing methods and to be overridden by other existing methods. For example, suppose `C` is a subclass of `B`, which is a subclass of `A`. An external operation could initially declare methods for receivers `C` and `A`, with a later glue method implementing the operation for `B`. However, the MultiJava implementation merges all the external methods declared in a single file into a single dispatcher class, with a single `apply` method, and this strategy does not allow "insertion" of methods into the middle of the specificity order. To fix this problem, RMJ compiles each method of an external operation into its own independent dispatcher class with its own `apply` method. In this way, all methods of external operations are treated as if they are glue methods, for the purposes of compilation.

3.2.2 Bytecode Annotations

The compiler must inform the class loader whenever load-time completeness or ambiguity checking is required for an operation. In that case, the compiler must additionally provide the loader with information about the methods declared on that operation, to enable the checking to be performed. Both of these tasks are accomplished via *method annotations*. The RMJ compiler produces a method annotation for each method in the program that belongs to an external operation. Each method annotation indicates the operation that the method is part of, the receiver and argument specialists (if any), the fully qualified names of its anchor and dispatcher classes, and whether or not the method is abstract. The annotation for a method declared in the file introducing the external operation is placed in the operation's anchor class bytecode. The annotation for an internal method added to the external operation is placed in the bytecode for the new method's receiver. Finally, the annotation for a glue method is placed in the associated glue anchor class bytecode.

Method annotations provide enough information for the loader to perform the necessary checking on external operations. For example, if a method annotation for an abstract external method is observed, then the loader will know to perform completeness checking. This checking relies on the other method annotations of the operation being checked, to decide whether the operation is fully implemented. Similarly, the appearance of a method annotation for a glue method or a method that specializes on an interface indicates that the associated operation requires ambiguity checking.

Method annotations are also generated for methods of regular internal operations, in order for the loader to check their completeness and ambiguity if necessary. It would be sufficient to generate an annotation for each method in the program, but this would be a large number of annotations. Worse, it wouldn't allow existing class files compiled by a regular Java (or MultiJava)

compiler to be used seamlessly (e.g., subclassed from) in RMJ programs.

Fortunately, the RMJ compiler can safely generate annotations for methods of internal operations on demand. First, if a concrete class does not declare or inherit a singly dispatched method for some operation, we generate a method annotation for the implicit abstract singly dispatched method. This alerts the class loader that completeness checking is necessary. We also generate method annotations for the multimethods on this operation declared in the current class. These annotations are sufficient for the loader to safely and precisely check completeness.

Second, the RMJ compiler must generate annotations to allow ambiguity checking of an internal operation whose methods specialize on interfaces. When such a method is observed, method annotations are created for it and for all other methods on its operation declared in the current class. For proper ambiguity checking of the operation, annotations are also needed for all methods of the operation in any subclasses and the superclass of the current class. Therefore, the existence of the method specializing on an interface triggers the compiler to generate appropriate method annotations in each subclass when it is compiled. However, the superclass has already been compiled, so it will in general not contain such annotations. Instead, we include the annotations for superclass methods in the bytecode for the current class. In this way, we generate the proper method annotations to enable load-time ambiguity checking, without either requiring existing code to be recompiled or generating method annotations for operations that do not require load-time checking.

3.3 Load-Time Extensions in RMJ

RMJ uses a custom class loader, named `RMJClassLoader`, that subclasses Java's standard `ClassLoader` class [Gosling et al. 00], to load the classes used in an RMJ program. This class loader observes each class loaded into the program and examines it for RMJ annotations.

The RMJ class loader is invoked in the following manner:

```
% java -Drmj.glue=<glue> RMJClassLoader  
    <Main> <args>
```

As described earlier, the class loader accepts a list of the glue files to be included in the current program; this is set via the `rmj.glue` property. Glue methods are processed in two phases: the first phase *registers* the existence of a glue method, and the second phase *loads* the glue method's dispatcher class and checks for redundant or ambiguously defined glue methods. The loader performs the first phase immediately, using the method annotations in the files named in the `rmj.glue` property. Each glue method is not actually loaded until it becomes reachable: its operation's anchor class, receiver, and argument specialists have been loaded. This strategy ensures that each glue method is only loaded if necessary and that it gets inserted in the appropriate place in the chain of dispatchers. Details on registering and loading glue methods are provided in Section 3.3.2.

Once all the glue is registered, the loader starts the RMJ program by loading the `<Main>` class and invoking its `main` method with the given `<args>`. `RMJClassLoader` will be the defining class loader [Liang & Bracha 98] for `<Main>`, which means that any classes referenced from that class will also be loaded with `RMJClassLoader`, transitively.

The key method of `RMJClassLoader` is `loadClass`, which takes the fully qualified name of a class to load, finds the bytecode implementation of the class, performs necessary RMJ checks on the

```

Class loadClass(String fullName) {
    Class c;
    if (fullName.startsWith("java.")) {
        c = findSystemClass(fullName);
        registerSuperclasses(c);
    } else {
        String fileName = asFileName(fullName);
        URL url = getResource(fileName);
        byte[] bytes = ..read contents of url..;
        c = defineClass(bytes);
    }

    registerClass(c);
    loadReachableMethods(c);
    verifyCompleteness(c);
    return c;
}

```

Figure 12: RMJClassLoader’s loadClass method

class, and creates and returns the `Class` object representing the loaded class. (This same process applies to interfaces as well. From the perspective of the virtual machine, interfaces are simply a special kind of abstract class. We adopt this perspective throughout the rest of this section, referring to both classes and interfaces generically as classes.) The overall procedure of RMJClassLoader’s `loadClass` method is sketched in Figure 12.

RMJClassLoader cannot be the defining loader for system classes, or else the classes will not be able to be passed to system methods. In the current implementation of RMJClassLoader, any class in the `java` package is loaded by the regular system class loader. Otherwise, we use the normal system mechanisms to find the class’s bytes. The class is then installed in the JVM using the inherited `defineClass` method, with RMJClassLoader as the defining loader, returning the new class object. The boldface operations in the `loadClass` method support RMJ’s load-time checking and are described in the rest of this section.

Java’s custom class loader mechanisms have enabled us to include additional load-time checking in the Java virtual machine. However, custom class loaders were intended to support multiple namespaces, not as a way for language designers to implement language extensions [Bracha 03], and they do not gracefully support all that we and other language designers might like. For example, custom class loaders for different extensions cannot be composed nicely. We view the design of a more flexible mechanism in Java for composable load-time checkers and code transformers to be an interesting area for future work.

3.3.1 Registering Classes

In order to perform completeness and ambiguity checking incrementally as classes are loaded, the loader maintains a number of data structures, which are described as needed in this section. The `registerClass` method updates these data structures appropriately whenever a new class is loaded. Aside from registering the existence of the new class, `registerClass` also reads any method annotations in the class and updates the data structures to reflect their existence. As mentioned earlier, all methods of external operations are treated as if they are glue methods for the purposes of compilation. Therefore, when `registerClass` finds an annotation for a method added to an

external operation, either a method in the file introducing the operation or a method in a subclass of the receiver, it registers the method exactly as glue methods from the `rmj.glue` property are registered. The methods will be loaded as they become reachable.

The RMJ class loader will not be the defining class loader for a system class. Consequently, classes referenced by the system class, such as its ancestor classes, may not be observed by the RMJ class loader. To partially account for this omission, the `registerSuperclasses` method calls `registerClass` on each of a system class’s superclasses, allowing the RMJ class loader’s data structures to reflect their existence. However, it is still possible for some relevant system classes to be missed, which can cause the loader to perform fewer checks than necessary to ensure correctness. An improved composable class loader mechanism would provide a way for custom class loaders to at least observe that these internal system classes have been loaded.

3.3.2 Registering and Loading Methods of External Operations

As described above, when an annotation for a method belonging to an external operation is found, either via the `rmj.glue` property or in some class, that method is registered. Registration consists in the creation of an *external method descriptor* for the method, which includes the fully qualified names of the external method’s anchor class, dispatcher class, receiver class, and argument specialization classes (or the static type of an argument, if it is unspecialized).

The new method will not be loaded until it is reachable. Therefore, the loader maintains an *external method registry*, which maps not-yet-loaded anchor, receiver, and argument specialization class names to the external method descriptors that are awaiting their loading. As part of a method’s registration, the registry is updated to reflect the classes upon which the new method is waiting. Finally, to speed external method loading (described next), each external method descriptor also stores a count of the number of distinct not-yet-loaded classes that it is waiting on. For example, when the glue method in Figure 8 is registered, it initially is waiting for the `area` operation’s anchor class and the `Triangle` class, assuming neither class has yet been loaded. The external method registry is therefore updated to reflect these dependencies, and the method’s descriptor gets a count of two.

The `registerClass` method, described earlier, is responsible for updating the external method registry to reflect the loading of a new class. That is, any mappings from the new class’s name in the registry are removed, and the mapped-to external method descriptors have their counts decremented. The `loadReachableMethods` operation then loads any method that has now become reachable, i.e. whose associated descriptor’s count is zero. Before loading the method, it is checked for unambiguity, as described later. Multiple methods of an external operation can become reachable simultaneously. In that case, `loadReachableMethods` loads the dispatcher classes of less-specific external methods and prepends them to the operation’s dispatcher chain *before* those of more-specific external methods, to ensure that overriding methods are always in front of their overridden methods on the chain.

3.3.3 Verifying Completeness

The `verifyCompleteness` method is used to ensure that operations remain complete in the face of abstract external methods and concrete classes that implicitly contain abstract singly dispatched methods. The loader must ensure that, for each such abstract method, for each tuple of concrete receiver and argument

classes that conforms to the abstract method's type signature, the abstract method is overridden by some loaded concrete method that is applicable to the tuple. To reduce the load-time work that is performed, only tuples consisting of *top concrete classes* of the abstract method's receiver and argument types need be considered. A concrete class C is a top concrete class of an abstract class D if there is no concrete class E (possibly identical to D) that is a superclass of C and a (possibly reflexive) subclass of D .⁴ If an operation has an incompleteness, it will be revealed by a tuple of top concrete classes. By similar reasoning, only *top concrete methods* of the abstract method need to be considered for applicability to these tuples. A top concrete method is a concrete method that directly overrides the abstract method, without any intervening overriding concrete methods.

Completeness checking in `verifyCompleteness` uses an incremental algorithm that works as each abstract method annotation and concrete class is loaded, without any redundant checking. When a new abstract method is loaded that needs completeness checking, the loader constructs all the conforming tuples of top concrete classes, based on the set of classes currently loaded, and checks that each has an applicable loaded method that overrides the abstract method. When a new concrete class C is loaded, the loader finds all loaded abstract methods that need completeness checking and have a receiver or argument type for which C is a top concrete class. For each such abstract method, the loader constructs all tuples of top concrete classes that contain C in some position and ensures that each has an applicable loaded method that overrides the abstract method.

For example, suppose the `area` methods in Figure 6 are loaded in an RMJ program. Assuming the `Shape`, `Rectangle`, and `Circle` classes have already been loaded, `verifyCompleteness` will check for the existence of `area` methods applicable to `Rectangle` and `Circle`, as each is a top concrete class of `Shape`. The method annotations in the `area` operation's anchor class allow this checking to succeed. When the `Triangle` class is later loaded, `verifyCompleteness` will check for the existence of an `area` method applicable to `Triangle`. If there is a subclass `area` method for `Triangle`, as in Figure 5, or a glue method for `Triangle`, as in Figure 8, it will have already been loaded by `loadReachableMethods` and will therefore be properly accounted for.

Our incremental completeness algorithm resembles the Rapid Type Analysis algorithm [Bacon & Sweeney 96]. Both algorithms maintain information about a set of reachable classes and a set of reachable operations. Whenever either set is extended, the new element is checked against all the existing elements of the other set. The algorithm is guaranteed at every point in time to have checked all pairs in the cartesian product of the two sets, without any duplicate checking.

The class loader maintains several data structures to make the checking of `verifyCompleteness` efficient. They are updated incrementally by `registerClass` as each class is loaded. The data structures are as follows:

- a mapping from each loaded abstract class to its set of loaded top concrete subclasses
- a mapping from each loaded abstract method needing completeness checking to its set of top concrete methods
- a mapping from each loaded abstract class to the set of loaded abstract methods needing completeness checking whose

4. Recall that throughout this subsection we are treating interfaces as special kinds of abstract classes.

receiver and argument types include the abstract class

For maximum flexibility, our `verifyCompleteness` implementation treats a completeness error as a non-fatal warning, and still allows the program to continue execution. If the incomplete scenario ever occurs at run time, then the exception that was compiled as the body of the abstract method will be thrown. It would be straightforward to parameterize the loader to allow different ways of treating load-time errors.

3.3.4 Verifying Unambiguity

As with completeness checking, the loader performs ambiguity checking on an operation incrementally, as each class is loaded. The heart of the loader's algorithm for incremental ambiguity checking is a routine that checks a pair of methods for ambiguity with one another. This algorithm can be used equally well to perform ambiguity checking at compile time, on the visible methods of an operation [Millstein et al. 02]. First, the receiver and argument specializers (C_1, \dots, C_n) and (D_1, \dots, D_n) for each of the two methods are retrieved. For now, we assume that the receivers and specializers are all classes; the generalization to interfaces is presented below. The algorithm checks several cases:

- If $(C_1, \dots, C_n) = (D_1, \dots, D_n)$, then the two methods are duplicates, and an ambiguity error is reported.
- Else if each C_i inherits from the corresponding D_i , then the first method overrides the second, and the methods are not ambiguous.
- Else if each D_i inherits from the corresponding C_i , then the second method overrides the first, and the methods are not ambiguous.
- Else if for each i , C_i and D_i are *related*, meaning that one inherits from the other, then the two methods may be ambiguous, because they are applicable to overlapping sets of argument tuples. This overlap is succinctly characterized by their *intersection tuple* $(\text{int}(C_1, D_1), \dots, \text{int}(C_n, D_n))$, where $\text{int}(C_i, D_i)$ returns whichever of C_i or D_i inherits from the other. The methods' overlap is not a problem as long as there exists a third method that is applicable to the intersection tuple and overrides the original two methods: the third method *resolves* the ambiguity of the first two. If such a method has been loaded, then the original two methods are unambiguous, and otherwise an ambiguity error is reported.
- Else the methods are *disjoint*: they are applicable to disjoint sets of argument tuples, and so they are unambiguous.

As a simple example, consider the `area` methods in Figure 4. All three pairs of methods pass the above check. The methods for `Rectangle` and `Circle` are disjoint from one another, because neither receiver inherits from the other. Further, each of these methods overrides the method for `Shape`. To illustrate intersection tuples, suppose that the `Shape` class of Figure 3 contained a method for drawing black-and-white printers:

```
public void draw(OutputDevice@BWPrinter p)
{ ... code for drawing a Shape on a b&w printer ... }
```

The first `draw` method of `Rectangle` in Figure 9 overlaps with the above method, and the intersection tuple is `(Rectangle, BWPrinter)`. Without the second `draw` method in `Rectangle`, whose receiver and argument specializer form exactly the intersection tuple, the original two methods would cause an ambiguity error to occur when `draw` is invoked on the intersection tuple.

As discussed in Section 2, an operation must undergo load-time ambiguity checking if either the operation has glue methods or has

methods that specialize on interfaces. We discuss each situation in turn.

3.3.4.1 Glue Methods

The loader records the set of methods that have been loaded for each operation. Then, just before `loadReachableMethods` loads the dispatcher class for a method belonging to an external operation, the new method is checked for ambiguity against each of the previously loaded methods with which it may be ambiguous, using the algorithm described above. It would be conservative for the loader to check the new method for ambiguity against each of the previously loaded methods. However, there is no need to recheck a pair of methods for ambiguity if their unambiguity was already established at modular compile time by the RMJ compiler. Any pair of methods that were simultaneously visible by the RMJ compiler during its compile-time checks on some file need not be rechecked at load time. Therefore, the only load-time checking that is required is between pairs of methods where one method is a glue method and the other method is either another glue method or a method written inside a class (such as the `area` method in Figure 5).

To exploit this observation, each external operation's list of previously loaded methods is partitioned into three separate lists, based on whether the method came from the source file introducing the external operation (a *base method*), the source file of a subclass that added a method to the external operation (a *subclass method*), or a glue method source file (a *glue method*); the method's annotation indicates which category the method is in. Whenever a glue method is loaded by `loadReachableMethods`, it is checked for ambiguity with those methods on the glue and subclass lists. Whenever a subclass method is loaded, it is checked for ambiguity with those methods on the glue list. No other combinations need load-time checking. In this way, operations with no glue methods will incur no load-time ambiguity checking.

3.3.4.2 Interface Specializers

Each operation containing methods that specialize on interfaces must be checked for unambiguity at load-time. To do so, we first generalize the routine described above for checking pairwise ambiguity of methods, to properly handle multiple inheritance. In particular, we now take into account the fact that two interfaces (or one interface and one class) can have a common subclass without themselves being related. Only the second-to-last case in the earlier routine needs to be modified. First, the case should apply when for each i , either C_i and D_i are related, as before, or there exists a loaded concrete class that inherits from both C_i and D_i . In the latter case, we define $\text{int}(C_i, D_i)$ to be the set of loaded concrete classes that inherits from both C_i and D_i . Finally, there are now multiple intersection tuples, each requiring a loaded resolving method, formed by taking the n -way cartesian product of the $\text{int}(C_i, D_i)$ sets.

The revised routine depends both on the set of currently loaded methods (in order to find resolving methods) and on the set of currently loaded concrete classes (in order to compute $\text{int}(C_i, D_i)$). Similar to incremental completeness checking, the loader must therefore incrementally check unambiguity of an operation containing methods with interface specializers as each of these sets changes. If the operation is external, unambiguity of a new method with respect to previously loaded methods is checked by `loadReachableMethods`, before the new method is loaded. If the operation is internal, unambiguity of a new method is checked when the method's annotation is found by `registerClass`. Finally, the custom class loader maintains a mapping from loaded interfaces to the loaded methods that specialize (either at the

receiver or an argument) on that interface; this mapping is updated by `registerClass` as classes are loaded. When a concrete class that implements an interface is loaded and registered, each method on the interface's list is retrieved and rechecked for ambiguity with respect to the other loaded methods of its operation.

As with the ambiguity checking of operations containing glue methods, we can optimize which pairs of methods need to be checked for operations containing interface-specializing methods. Only pairs of methods where at least one has an interface specializer need to be checked for ambiguity by the RMJ loader.⁵ All other pairs are guaranteed to be unambiguous because of the RMJ compiler's modular checks. For external interface-specializing operations, our optimization requires that a method specializing on an interface be checked against *all* previously loaded methods, including base methods. Although each method was checked against the base methods modularly by the RMJ compiler, the compile-time checks may have missed ambiguities caused by unseen concrete classes that inherit from interfaces.

3.3.4.3 Run-time Ambiguity Checking

As with completeness errors, to give programmers increased flexibility, ambiguity errors are treated by our RMJ class loader as non-fatal warnings, and the program is allowed to continue. Whenever a load-time ambiguity error is reported for some operation, a special ambiguity dispatcher class, whose `apply` method throws a `RuntimeException`, is instantiated and prepended to the operation's dispatcher list. If the ambiguity is caused by duplicate methods, then the ambiguity dispatcher's `apply` method has the same receiver and argument specializers as each of the duplicates. If the ambiguity is caused by the lack of a method for an intersection tuple, then the ambiguity dispatcher's `apply` method has the same receiver and argument specializers as the intersection tuple. In this way, the ambiguity dispatchers ensure that an exception will be thrown whenever a run-time ambiguity occurs. This design only works for external operations; if the load-time ambiguity error for an internal operation is ignored by the programmer and the ambiguity is encountered at run time, one of the ambiguously defined methods will be invoked arbitrarily.

3.4 RMJ Preloader

When developing an application in RMJ, the programmer may wish to exploit expressiveness that cannot be checked purely modularly, at the cost of taking on the responsibility of avoiding load-time incompletenesses and ambiguities. The RMJ class loader will check for these problems when the program is run on *one* particular input. However, it would also be useful to know whether or not a program can incur load-time errors at all, for *any* possible input. For example, a developer of shrink-wrapped software might wish to verify, once and for all, that no load-time errors can occur for a program. As another example, a programmer integrating two libraries may wish to find places where those libraries require glue in order to avoid load-time errors.

To assist in this kind of checking, we have developed a "preloader" tool. The preloader is invoked like the custom class loader, except that no arguments are given:

```
% java java -Drmj.glue=<glue> RMJPreLoader
    <Main>
```

5. For an operation that has both glue methods and interface-specializing methods, the loader must check any pair that is not eliminated by *both* optimizations.

The preloader starts by registering all the glue listed in the `rmj.glue` property, just as `RMJClassLoader` does. The preloader then exhaustively explores all classes transitively referenced from the `<Main>` class, ignoring the application's actual flow of control. The preloader performs all the RMJ load-time checks as it visits each class. Even though classes may be visited by the preloader in a different order than in a real execution, and more classes may be visited by the preloader than in a real execution, the preloader is guaranteed to discover all potential load-time hazards of a real execution, with one caveat described below. If the preloader reports a hazard, then the programmer is given early warning about a situation needing attention. If the preloader reports no hazards, then the programmer has increased confidence that the program will work correctly at run time.

The preloader's one caveat is that it only considers statically referenced classes. It will not examine classes loaded only through reflective mechanisms such as `Class.forName`. Many applications only reference classes statically, and most others only rarely reference classes through reflection, so this limitation should not greatly hinder the preloader's accuracy. Further, the places in a program where this limitation arises are clear by simply scanning the source text (or bytecodes).

3.5 Discussion

Several interesting issues arise in the context of load-time typechecking. RMJ's load-time checking has been designed to work in an incremental fashion, in order to fit nicely in the context of Java's support for lazy class loading. The Java language does not mandate that such an on-demand loading strategy be used. For example, an implementation is free to load several classes as a group or prefetch classes for later use [Gosling-etal00]. The RMJ class loader will function properly regardless of an implementation's loading strategy. However, because different Java implementations may load classes in different orders, they may also trigger RMJ's load-time errors in different orders.

One way to resolve this problem is to make use of the fact that Java mandates fairly precisely when a class must be initialized. Effectively, a class must be initialized immediately before its first use. Therefore, we could modify our implementation of RMJ to perform each class's "load-time" checks at *initialization* time rather than at load time. Instead of using a custom class loader, the RMJ compiler would add appropriate calls from each class's static initializer to the routines for performing RMJ's completeness and ambiguity checks.

The RMJ class loader trusts the method annotations inserted into class files by the RMJ compiler. If the annotations are inaccurate, the loader may miss real message dispatch errors or signal spurious ones. However, only dispatch errors as defined by RMJ are potentially compromised by this trust relationship. Importantly, the regular Java bytecode verification process is unaffected: the Java verifier can independently check Java's well-formedness conditions on each class file, without requiring any trust in the RMJ compiler. Therefore, a running RMJ program is guaranteed not to violate the integrity of the underlying Java virtual machine.

It is possible that the loader could be augmented to independently verify that a class adheres to its associated method annotations, because of the stylized way in which external methods and multimethods are compiled. For example, a method containing a sequence of `instanceof` tests could be checked to correspond to a given set of annotations for multimethods. However, it would be very difficult to detect missing method annotations, because RMJ code compiles to regular Java bytecode. For example, it would be

impossible to know whether the `instanceof` tests in some bytecode method were produced via translation from RMJ's multimethods or were instead written directly by the original Java programmer.

4. STATUS AND EXPERIENCE

We have developed an implementation of RMJ. We extended the MultiJava compiler to handle the additional RMJ language features and code generation strategy, and we implemented `RMJClassLoader`, `RMJPreloader`, and associated helper classes. Our implementation is freely available for download as a part of the regular MultiJava system [Mul]. The implementation includes all of the features and algorithms described in this paper, with one exception. Due to complications in the design of the Java compiler underlying MultiJava's compiler, it is challenging to allow concrete classes lacking appropriate singly dispatched default methods (e.g., the example in Figure 10), so we do not currently support this idiom.

The next subsection describes a larger example using RMJ than the ones presented so far, and the second subsection reports on some performance experiments using this example.

4.1 A Case Study

We have experimented with rewriting in RMJ parts of Barat [Bokowski & Spiegel 98], a Java front-end written by others. Barat builds an abstract syntax tree (AST) from a set of Java source files, which can then be used to perform various static analyses over the given code. Barat is itself written in Java, and the AST nodes are represented by a class hierarchy, with root interface `Node`. Barat uses the visitor design pattern in order for clients to perform their desired analyses without modifying the node classes directly. To write an analysis, clients create a new class implementing the `Visitor` interface. To invoke the analysis, clients invoke an AST node's `accept` method, passing an instance of the new visitor.

Barat comes with several predefined visitors. One of them, the `OutputVisitor`, outputs a source code representation of the given AST nodes. We re-implemented this functionality using RMJ, by writing an external operation, `output`, on the AST node classes. As opposed to the visitor pattern, which requires "hooks" (the `accept` method) inside the node classes, the implementer of Barat did not need to plan ahead to allow us to implement our revised output operation. Further, it was natural to define the output operation to take parameters, for example the current indentation and the stream to which the output should be directed. Because the `OutputVisitor` has to conform to the `Visitor` interface, these parameters must instead be simulated via fields in the `OutputVisitor` class. Finally, clients can invoke the output operation via ordinary message sending syntax, as if it were defined in the original node classes.

Those benefits would be obtained via an `output` external operation in regular MultiJava, but the output operation also benefits from the new features of RMJ. In MultiJava, the output operation would be forced to contain a default implementation for the root interface `Node`, to handle any unseen concrete subclasses. However, there is no reasonable default behavior in this case, so the default method would be forced to simply throw a run-time exception. The presence of the default implementation also means that static exhaustiveness checking succeeds trivially, even though the intent is that the default method should never be invoked. In RMJ, `output` contains an abstract method for `Node`. During modular static typechecking, a warning is signaled, and any visible subclasses are checked for exhaustiveness. Our custom class loader

then checks at load time to ensure that all subclasses of `Node` do indeed have an appropriate implementation of `output`.

The `output` operation also naturally employs RMJ's glue methods. One way Barat has been used is to experiment with extensions to Java (e.g. [Aldrich et al. 02]). Clients add their own subclasses of `Node` to represent the new syntax and update the Barat parser appropriately. Unfortunately, the client extensions break all existing visitors, which do not know how to visit the new nodes. If clients wish to use the `OutputVisitor`, it must first be modified in place to contain methods for visiting the new nodes, and then retypechecked and recompiled. Using MultiJava's external methods instead of visitors would suffer from a similar problem: because all external methods of an operation must be in the same file, source access to the `output` operation would be required in order to add `output` external methods for the new nodes.

In RMJ, the `output` operation can be updated to handle the new nodes, without requiring source access to the original `output` external methods, or even recompiling them. We simply create a new file containing glue methods that provide `output` functionality for the new nodes. As an example, we created a version of the `output` operation that does not support Barat's `Cast` and `InstanceOf` nodes, representing Java's run-time `cast` and `instanceof` test, respectively. Therefore, the `output` operation is only well-defined on the subset of Java programs that do not perform explicit run-time type manipulation. To handle the "extension" allowing casts and `instanceof` tests, we then created two `output` glue methods handling the new nodes, without modifying the original `output` code or the new nodes. Clients of `output` whose Java programs employ run-time type manipulation add the glue files to their `rmj.glue` property to make the two independent extensions to the `Node` hierarchy (the `output` operation and the new node subclasses) work together.

A final use of RMJ's expressiveness is required by Barat's use of interfaces as the sole external view of its functionality. Barat provides its AST nodes in two parallel hierarchies: as a set of interfaces, and as an associated set of classes implementing those interfaces. The intent is that clients never interact directly with the implementation classes, but only with the interfaces. `Node`, `Cast`, `InstanceOf`, and all other `Node` kinds are public Java interfaces; internal concrete classes like `CastImpl` and `InstanceOfImpl` implement these public interfaces. RMJ allows the various `output` methods to be defined directly on the public interfaces and ensures that there are no multiple-inheritance ambiguities at load time.

One benefit of the original visitor implementation is that it can be inherited for use by other visitors. For example, a `LoggingVisitor` could subclass from `OutputVisitor` and override a few of the `visit` methods to print some extra logging information, while inheriting the rest of the `visit` methods. Writing a logging external operation that forwards to our `output` operation would not work, since recursive calls would all go to `output` instead of back to the logging operation.

If inheritance of visitors is desired, an alternative strategy in RMJ is to implement the `OutputVisitor` as an `Output` class that contains an operation accepting the node being visited as an argument, as shown in the top of Figure 13. When the `apply` operation is invoked, multimethod dispatch is used to provide the appropriate implementation for each node. The `Output` class can then be extended by a `Logging` class, analogous to the `LoggingVisitor`. But unlike the visitor-based approach, the `Output` class using multimethods requires no advance planning from the implementer of the node hierarchy. Additionally, `apply`

```
public class Output {
    public void apply(Node@ifNode n) {
        ... code for outputting an if statement ...
    }
    public void apply(Node@whileNode n) {
        ... code for outputting a while statement ...
    }
    ...
}

-----
public void Node.output() {
    new Output().apply(this);
}
```

Figure 13: An alternative to visitors in RMJ

multimethods within the `Output` class can inherit from one another, unlike the various `visitX` methods of the `OutputVisitor` class. Finally, an external operation named `output` can be written as a wrapper around a call to `Output`'s `apply` method, as shown in the bottom of Figure 13, so that clients can use their normal calling sequence to invoke the operation.

4.2 Performance Experiments

RMJ adds run-time overhead to perform its load-time checking. To gauge the performance cost of RMJ's specialized class loader, we ran four different versions of the `output` functionality described above. The first version is the original `OutputVisitor` class provided with Barat. The second is an external `output` operation using regular MultiJava. Because of MultiJava's restrictions for modular safety, this version includes a concrete default implementation for `Node`. In addition, all of the `output` methods are declared in a single file, and they are defined directly on the internal classes (e.g., `CastImpl` and `InstanceOfImpl`) rather than the external interfaces (e.g., `Cast` and `InstanceOf`). The third version is an RMJ version of the external `output` operation, which uses an abstract method for `Node` and uses glue methods for the `output` methods for casts and `instanceof` tests, but it leaves the `output` methods defined on the internal classes. The fourth version is the "ideal" RMJ version, which is like the third version except that the `output` methods are defined on the external interfaces. The first two versions can be run with either Java's regular class loader or with the RMJ class loader, but the third and fourth versions can only be run using RMJ's custom loader.

We invoked each version of the `output` functionality on two inputs: a small input that's a single Java source file 662 lines in length, and a large input that's 20 Java source files 7476 total lines in length. Barat parses the file(s), creates the associated AST nodes, and then invokes the `output` functionality (either the `OutputVisitor` or the `output` operation) to print out the source-code representation of the nodes. All reported times are the median value of the user time of five runs, measured on a SunBlade 1000 Model 2900 with 5GB RAM running SunOS 5.8 and Sun Java SDK1.4.1. Table 1 presents the results of these experiments.

The "passive" overhead for using the RMJ class loader instead of the regular Java class loader, when the RMJ class loader has no load-time checks to perform (the difference between the two loaders for each of the first two versions), is 8-9% for the small input and to 3-8% for the larger input. When the RMJ class loader has to do work to load external methods (including glue methods) and to check for load-time incompleteness and ambiguities (the difference between the MultiJava version using the Java loader and

Table 1: Performance

input size	version	loader	time (secs)	over-head
small	Java OutputVisitor	Java	3.9	0%
		RMJ	4.2	8%
	MultiJava output on classes	Java	4.1	6%
		RMJ	4.4	15%
	RMJ output on classes	RMJ	4.3	11%
	RMJ output on interfaces	RMJ	4.6	20%
large	Java OutputVisitor	Java	9.8	0%
		RMJ	10.5	8%
	MultiJava output on classes	Java	10.3	6%
		RMJ	10.6	9%
	RMJ output on classes	RMJ	11.2	14%
	RMJ output on interfaces	RMJ	10.9	12%

each of the two RMJ versions), the overhead becomes 5-14% for the small input and 6-8% for the larger input. Because the overhead of RMJ's class loader is incurred only when a class is loaded, we would expect that as programs run longer, the impact of the load-time checks on performance becomes less important. This expectation is borne out by these results, although the differences in times are so small that the run-to-run variations in times generally are greater than the measured overhead for RMJ.

The overhead for using external methods instead of visitors (the difference between the Java version and the MultiJava version) is 1-7%. This cost is incurred throughout execution, not at load time, and is due to the inefficient implementation of external operations as compared with the implementation of ordinary Java methods. This cost is largely independent of whether MultiJava or RMJ is used.

5. PREVIOUS WORK

The inspiration for the features in MultiJava and RMJ comes from previous languages based on multimethods, including CLOS [Steele Jr. 90, Paepcke 93], Dylan [Shalit 96], and Cecil [Chambers 92, Chambers 93]. These languages support arbitrary multimethods and external methods (indeed, all methods are written external to their classes). However, CLOS and Dylan are dynamically typed, and Cecil requires global typechecking to ensure type safety of message sends [Litvinov 98]. Vortex [Dean et al. 96], the compiler for Cecil (and other languages), employs a global compilation strategy that makes heavy use of whole-program optimization.

Parasitic methods [Boyland & Castagna 97] and Half & Half [Baumgartner et al. 02] are both extensions to Java that support *encapsulated* multimethods [Castagna 95], which are akin to internal multimethods in RMJ; neither language supports external (multi)methods. Like RMJ, both languages support the use of

interfaces as specializers in multimethods. Because it is difficult to modularly check multimethod ambiguity in the presence of interface specializers, parasitic methods modify the multimethod dispatch semantics so that ambiguities cannot exist, employing the textual order of methods to break ties. Half & Half resolves the problem by performing ambiguity checking on entire packages at a time, rather than on individual classes. For such package-level checking to be safe, Half & Half must also limit the visibility of some interfaces to their associated packages, thereby disallowing outside clients from employing them as specializers. In contrast, RMJ ensures that operations are unambiguous without either modifying the multimethod dispatching semantics or imposing restrictions on the usage of interface specializers. Instead, RMJ requires incremental ambiguity checking at class load time.

Recently several languages have emerged that provide direct support for separation of concerns. For example, AspectJ [Kiczales et al. 01] is an aspect-oriented extension to Java, whose *aspects* can extend existing classes in powerful ways. HyperJ [Ossher & Tarr 00] is a subject-oriented extension to Java that provides *hyperslices*, which are fine-grained modular units that are composed to form classes. Both languages support open classes; for example, this ability corresponds to AspectJ's *introduction* methods. The languages additionally support many more flexible extensibility mechanisms than RMJ. For example, AspectJ's *before* and *after* methods provide ways of modifying existing methods externally. To cope with this level of expressiveness, these languages employ non-modular typechecking and compilation strategies. For example, AspectJ's compiler *weaves* the aspects into their associated classes; only when all aspects that can possibly affect the class are available for weaving can typechecking and compilation be completed.

Binary Component Adaptation (BCA) [Keller & Hölzle 98] allows programmers to define *adaptation specifications* for their classes, which can include the addition of new methods, thereby supporting open classes. Adaptation specifications can also include modifications not supported by RMJ, like the declaration that an existing class implements some particular interface. The typechecking and compilation strategy is similar to the aspect weaving approach described above, requiring access to all adaptation specifications that can affect a given class in order to typecheck and compile the class. The authors describe an on-line implementation of BCA, whereby the weaving is performed dynamically using a specialized class loader. Our strategy is superficially similar, with glue methods playing the role of adaptation specifications. However, RMJ's class loader simply verifies that loaded classes are type-safe when this cannot be fully determined modularly. No modification to the class files is necessary, and a class can be safely verified and loaded before its augmentations are available.

Jiazzi [McDirmid et al. 01] is an extension to Java that provides components with a powerful external linking semantics, including recursive linking. The authors show how to use these features to encode an *open class pattern*, whereby a component imports a class and exports a version of that class modified to contain a new method or field. Open classes in RMJ (and MultiJava) allow two clients of a class to augment the class in independent ways, without having to be aware of one another. In contrast, in Jiazzi there must be a single component that integrates all augmentations, thereby creating the final version of the class. Component linking in Jiazzi is performed statically, so it is not possible to dynamically add new methods to existing classes. Dynamic augmentation is possible in RMJ, since it is integrated with Java's regular dynamic loading process.

The visitor design pattern [Gamma et al. 95] is a programming idiom that allows new operations to be added to existing classes without modifying existing code. However, the visitor pattern has several drawbacks that are not shared by open classes in RMJ, as discussed in Section 4.1. Most importantly, the ability to add new operations to an existing class comes at the cost of losing the ordinary object-oriented ability to add new subclasses, since each visitor must be modified to handle the new subclass. Several researchers have designed extended versions of the visitor pattern that resolve this issue [Krishnamurthi et al. 98, Palsberg & Jay 98, Martin 98, Nordberg 98, Vlissides 99, Zenger & Odersky 01]. However, these extensions require dynamic type casts or run-time reflection, and they often complicate the already-complex visitor pattern.

6. CONCLUSIONS AND FUTURE WORK

RMJ represents a new point in the design space balancing extensibility against modular reasoning. It offers almost the full power of external methods and multimethods while retaining all of MultiJava’s modular typechecking guarantees. Many of the supported idioms can be proved safe purely modularly, independent of how the module is used in enclosing programs, and the remainder are proved safe incrementally as each module is loaded. A preloader tool assists in discovering load-time errors before run time. Programmers can explicitly choose between expressiveness and early checking, based on their software development needs and goals.

Unlike some related systems that also offer greater extensibility, RMJ retains a modular approach to typechecking and compilation. RMJ can check modules separately, and either guarantee them safe or point out exactly those situations that programmers must be concerned about to avoid load-time errors. Current systems based on global or large-scale translation or weaving to combine separate concerns do not provide these kinds of early checking.

Much of the challenge in developing RMJ was in designing the interplay between compile-time and load-time checking and code generation, to keep load-time overhead small. RMJ’s implementation strategy performs all code generation at compile time in a modular, per-file fashion. It also attempts to perform as much checking modularly as it can. For those checks that lead to warning messages to programmers, additional annotations are generated, directing the load-time checker’s efforts to those parts of the program that need load-time checking. The loader maintains several data structures that help it to perform the needed checks efficiently.

In future work, we plan to pursue several directions:

- To gain experience with the strengths and limitations of RMJ, we will be using RMJ in the implementation of several large systems. We have already been using MultiJava in several case studies, and this experience was one motivation for designing RMJ.
- We wish to explore supporting additional extensibility while retaining modular or load-time checking. It would be useful to declare that a class implements an interface outside of the class (for example, along with adding external operations to the class). Adding static methods and static fields to a class from the outside would be simple but useful extensions. Adding instance fields and constructors to a class from the outside would also be useful but is more challenging to implement efficiently. We also wish to investigate how we might incorporate some of the additional extensibility of systems like AspectJ and Hyper/J, particularly the ability to extend

individual methods with additional “before” and “after” behavior from the outside, while retaining modular checking.

- We wish to investigate including binary code generation or rewriting as part of custom class loading. As mentioned in Section 3.1, we could dynamically generate more efficient dispatching methods for external operations. When a new dispatcher class is loaded, any previous dispatcher method would be invalidated and dropped. The next time the external operation is invoked, a customized dispatcher method based on the list of currently loaded dispatchers would be dynamically generated, loaded, and invoked. Previous efficient multimethod dispatching algorithms can be used when generating the dispatcher based on the current snapshot of loaded methods [Chambers & Chen 99]. Binary code generation could also be used to allow additional kinds of extensibility that are challenging to implement modularly, including the ability to write glue methods belonging to regular internal operations.

Acknowledgments

Thanks to Curtis Clifton, Gary Leavens, and the anonymous reviewers for helpful feedback on this work. We are grateful to Curtis Clifton, Gary Leavens, and the rest of the MultiJava team for their work on the MultiJava implementation, which formed the basis for the RMJ implementation. This work has been supported in part by NSF grants CCR-0204047 and CCR-9970986, NSF Young Investigator Award CCR-945776, and gifts from Sun Microsystems and IBM.

7. REFERENCES

- [Aldrich et al. 02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, LNCS 2374, Malaga, Spain, June 2002. Springer-Verlag.
- [Andersen & Reenskaug 92] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *Proceedings of the 1992 European Conference on Object-Oriented Programming* [ECO92], pages 133–152.
- [Bacon & Sweeney 96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP96].
- [Baumgartner et al. 02] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Department of Computer and Information Science, The Ohio State University, March 2002.
- [Bobrow et al. 86] Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP86], pages 17–29.
- [Bokowski & Spiegel 98] Boris Bokowski and Andre Spiegel. Barat — A Front-End for Java. Technical Report Technical Report B-98-09, Freie Universitat Berlin, FB Mathematik und Informatik, December 1998.
- [Boyland & Castagna 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, GA, October 1997.
- [Bracha 03] Gilad Bracha. Personal communication. January 2003.

- [Cartwright & Fagan 91] Robert Cartwright and Mike Fagan. Soft Typing. *SIGPLAN Notices*, 26(6):278–292, June 1991. Conference on Programming Language Design and Implementation.
- [Castagna 95] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.
- [Chambers & Chen 99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 238–255, Denver, CO, November 1999.
- [Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the 1992 European Conference on Object-Oriented Programming* [ECO92], pages 33–56.
- [Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering, University of Washington, March 1993. Revised, March 1997.
- [Chambers 98] Craig Chambers. Towards Diesel, a Next-Generation OO Language after Cecil. Invited talk, the *Fifth Workshop of Foundations of Object-Oriented Languages*, San Diego, California, January 1998.
- [Clifton 01] Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. Available from multijava.sourceforge.net.
- [Clifton et al. 00] Curtis Clifton, Gary Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Symmetric Multiple Dispatch and Open Classes for Java. In *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, MN, October 2000.
- [Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP96].
- [ECO92] *Proceedings of the 1992 European Conference on Object-Oriented Programming*, LNCS 615, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [ECO98] *Proceedings of the 1998 European Conference on Object-Oriented Programming*, LNCS 1445, Brussels, Belgium, July 1998. Springer-Verlag.
- [Gamma et al. 95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.
- [Gosling et al. 00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.
- [Harrison & Ossher 93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, D.C., October 1993.
- [Keller & Hölzle 98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.
- [Kiczales et al. 97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [Kiczales et al. 01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, LNCS 2072, Budapest, Hungary, June 2001. Springer-Verlag.
- [Krishnamurthi et al. 98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In *Proceedings of the 1998 European Conference on Object-Oriented Programming* [ECO98], pages 91–113.
- [Liang & Bracha 98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP98], pages 36–44.
- [Litvinov 98] Vassily Litvinov. Constraint-Based Polymorphism in Cecil: Towards a Practical and Static Type System. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP98].
- [Martin 98] Robert Martin. Acyclic visitor. In *Pattern Languages of Program Design 3*, pages 93–104. Addison-Wesley, 1998.
- [McDirmid et al. 01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa, FL, October 2001.
- [Millstein & Chambers 99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. In *Proceedings of the 1999 European Conference on Object-Oriented Programming*, LNCS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.
- [Millstein et al. 02] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 110–122, New York, NY, September 2002. ACM.
- [Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP86], pages 1–8.
- [Mul] MultiJava home page. <http://multijava.sourceforge.net>.
- [Nordberg 98] Martin E. Nordberg. Default and extrinsic visitor. In *Pattern Languages of Program Design 3*, pages 105–123. Addison-Wesley, 1998.
- [OOP86] *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, November 1986.
- [OOP96] *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, October 1996.
- [OOP98] *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, October 1998.
- [Ossher & Tarr 00] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737, June 2000.
- [Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.
- [Palsberg & Jay 98] Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 August 1998.
- [Shalit 96] Andrew Shalit. *The Dylan Reference Manual*. Addison-Wesley, Reading, MA, 1996.
- [Smaragdakis & Batory 98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the 1998 European Conference on Object-Oriented Programming* [ECO98], pages 550–570.

- [Steele Jr. 90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, second edition, 1990.
- [VanHilst & Notkin 96] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *Proceedings of 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.
- [Vlissides 99] John Vlissides. Visitor in Frameworks. *C++ Report*, 11(10):40–46, November/December 1999.
- [Zenger & Odersky 01] Matthias Zenger and Martin Odersky. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*. ACM, September 3-5 2001.