A Modular Compilation Strategy for Open Classes and Multimethods in Java

Gregory H. Cooper

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

April 30, 2002

[illegible]

Professor Shriram Krishnamurthi
Advisor

A Modular Compilation Strategy for Open Classes and Multimethods in Java

Gregory Cooper

Abstract
Open classes and multiple dispatch provide a powerful mechanism for constructing extensible object-oriented software. However, their use significantly complicates the task of performing modular type-checking and compilation. Existing implementations therefore make various trade-offs. For example, CLOS is dynamically typed, Cecil relies on global link-time checking, and MultiJava forbids the expression of many reasonable programs. In this paper, we present a new system that extends Java with open classes and multiple dispatch. Our system provides better expressiveness than MultiJava while preserving static type-safety and integrating naturally with Java's modular compilation mechanisms.

1 Introduction

As software systems become ever larger and more complex, the ability to reuse existing program fragments becomes increasingly important. Already, most programming languages provide module systems, which allow programmers to compile parts of a system separately and later link them together. A common form of software reuse involves libraries—compiled definitions of datatypes and functions whose distributors generally do not provide source code and which therefore cannot be modified.

While software libraries provide an important form of code reuse, programmers often require the ability to extend part of a library's functionality, such as by adding a new variant to a datatype or by defining a new operation over an existing datatype. For example, a set of libraries may implement parts of a development environment for a programming language, such as a parser, an interpreter, and a user interface. An independent programmer, who does not have access to the original source code, may wish to extend such an environment with a new linguistic construct (a variant in the abstract syntax tree datatype) or a new form of static analysis (an operation defined over the existing datatype).

These two forms of extensibility are orthogonal, and often both are desirable within the same appplication. Unfortunately, existing programming languages provide support for only one of the two. In particular, functional languages (such as ML or Haskell) allow programmers to define new operations over existing datatypes, but they do not permit the modular extensibility of the set of variants comprising a datatype. In contrast, object-oriented languages (like Java) allow programmers to define new variants of an existing datatype, but they do not permit modular extension of existing type hierarchies with new operations.

To overcome these limitations, programmers have developed design patterns that permit extensibility in the dimension that the language does not support directly. For example, in the context of object-oriented languages, the Visitor [4] pattern creates a framework for extending class hierarchies with new operations. Likewise, Steele [6] develops a "tower of types" pattern that effectively supports modularly extensible datatypes in a functional language.

Design patterns are an unsatisfactory solution to the problem of software extensibility. Importantly, library implementors may not foresee the need for extensions to their code, in which case they are unlikely to follow the necessary patterns. Furthermore, the patterns suffer from limitations. For example, the Visitor pattern allows operational extensions to a *fixed* type hierarchy; it cannot safely accommodate the introduction of new variants. Similarly, Steele's pattern builds an elaborate framework for extending datatypes in the presence of a fixed set of operations, but it is not clear how to generalize his technique to allow the addition of new operations.

2 Extensibility in Object-Oriented Languages

Recognizing the limitations of design patterns, researchers have investigated augmenting languages with features that support extensibility. In particular, other work [1, 2, 3, 5] has investigated the addition of open classes and multiple dispatch to object-oriented languages.

Open Classes

Open classes allow programmers to add new members to existing classes. To see why they are useful, imagine a programming environment like the one mentioned in the introduction. For clarity, assume that the application is written in Java. A class library provides a hierarchy of *Expression* classes, which represent program expressions and implement behaviors such as evaluation and pretty-printing. In addition, the library provides a parser that constructs *Expressions* from strings.

Suppose that we want to extend the *Expression* hierarchy with a type-inference operation. We might consider any number of strategies in attempting to achieve this goal, but none provides an adequate solution.

For instance, we might think about deriving a new subclass of *Expression* that contains the new operation. Unfortunately, in Java (as in many common object-oriented languages), a class's superclass is fixed during compilation. Thus, the programmer cannot modularly update existing subclasses to inherit from a new superclass.

For now, however, let us assume that we *can* "reparent" the subclasses in this way. Even with this assumption, writing down the code for the new method is troublesome, since it may depend upon the implementations of the specific subclasses on which it is invoked. With run-time type tests and downcasts, we can define all of these cases in a single method in the new superclass. However, if someone later adds a new subclass to the system, this method will fail, and the only way to fix it, in general, will be to edit the source code.

It seems that what we really want in such a case is the ability to put a new *abstract* method in the superclass. Then, the language can require that each subclass provide an appropriate implementation. Unfortunately, if we could do this, we would be stuck once again; without source code for the subclasses, we would have no way of implementing the new method in them. We could derive a new subclass for each existing subclass, but then the parser (which constructs all of our *Expressions*) would have no way of knowing about the new subclasses.

The preceding scenario exemplifies a general problem in modular software construction that cannot be solved naturally in existing object-oriented languages. With the addition of open classes, however, we can overcome all of the weaknesses mentioned above. We *can* add abstract methods to the root of the class hierarchy, and we can compile implementations directly into the concrete subclasses. Importantly, if we later try to add new variants to the class hierarchy, the type system will require that they implement the new method.

Multiple Dispatch

While open classes solve a large part of the object-oriented extensibility problem, they do not provide a complete solution. The reason is that datatype extensbility in object-oriented languages is largely due to the dynamic dispatch mechanism, which permits a form of open recursion. Dynamic dispatch, however, applies only to the target of a method invocation, but in some applications a method's behavior depends upon the dynamic types of *several* of its arguments. Many binary methods, such as equality, unification, and arithmetic operators, have this property.

Common object-oriented languages (like Smalltalk, Java, Eiffel, and C++) only provide single dispatch. Thus, programmers cannot express these sorts of methods naturally and must instead follow a pattern. Typically, this means performing dynamic type tests to determine the argument types and then downcasting the arguments to their runtime types before performing an appropriate sequence of operations. This technique can be tedious and error-prone. Furthermore, once such a method is written, a programmer without access to the source code cannot extend it to handle cases involving new variants of the argument types. If the programmer planned in advance to support extensibility, he could build a framework in which each argument class participated in the method selection process by providing a dispatcher. However, this pattern is also error-prone and can be extremely tedious if there are more than a few argument classes.

In order to support modular extensibility in a natural manner, we would like the language to provide a mechanism for *multiple dispatch.* In this case, instead of employing a cumbersome pattern, a programmer can implement a multiply-dispatched function simply by providing several method implementations and indicating which implementations apply to which types of arguments. We call such a function a "generic function" and the methods that comprise its implementation "multimethods". In a language supporting this style of programming, the runtime system automatically selects the most appropriate multimethod for each invocation of the generic function.

Code written to take advantage of multimethods has several advantages over the corresponding code written without the aid of multiple dispatch. In addition to being modularly extensible, the code is clearer and safer, since dynamic type tests and casts are subsumed by the dispatch mechanism. Multiple dispatch makes a language more expressive, a feature that programmers often desire.

Implications for Modular Compilation

The convenience of open classes and multiple dispatch do not come for free. In particular, they raise issues for modular type-checking and compilation.

In languages like Java, a given class definition depends upon a single compilation unit. With the addition of open classes, a single class definition may depend upon several compilation units. Without support for this capability in the language, it is not immediately obvious how to implement it. Abstract methods significantly complicate the situation since, not only can a programmer add operations to a class, he can enlarge the set of operations that *other* classes must implement. The combination of multimethods and open classes can make things even more complex. In particular, when we have methods defined over abstract arguments, we may choose not to require default implementations and instead only demand that there be an implementation for each tuple of *concrete* argument types. In this case, the introduction of a new, seemingly independent class into a system may result in additional constraints on existing classes. Clearly, such effects make modular compilation difficult.

Previous Work

Because of the issues involved in implementing open classes and multiple dispatch, previous efforts have involved a variety of trade-offs. For instance, early implementations of multiple dispatch, as in CLOS [1], are not statically typed at all. While this approach simplifies the implementation, many programmers are unwilling to sacrifice the safety guarantees afforded by a static type system.

Work on Cecil [2] explores the use of open classes and multiple dispatch in a statically-typed object-oriented language. Cecil permits unrestricted extensibility but, as a result, it needs to perform global type-checking and compilation.

Dubious [5] attempts to overcome Cecil's need for global type-checking and compilation, and MultiJava [3] applies the techniques developed for Dubious to an existing, practical programming language. However, in order to support modular type-checking and compilation in these languages, the designers place stern restrictions on the use of the features. In particular, they disallow the extension of existing type hierarchies with new abstract methods, and they limit the set of compilation units to which multimethods may belong. The effect is to prohibit the expression of some reasonable forms of extensibility, in particular those in which extensions to the sets of both operations and variants occur.

MultiJava

Our work is inspired by MultiJava and attempts to improve upon it by allowing more flexible extensibility without sacrificing modular type-checking or compilation. To make the distinctions between our work and MultiJava clearer, we provide a brief description of MultiJava here.

MultiJava extends Java with two new syntactic constructs, one for open classes and one for multiple dispatch. To use the open class mechanism, a programmer writes something like:

```
Type LambdaExpr.inferType(TypeEnv env) { ... }
```

This declaration lies outside of any class body and adds a method named *inferType* to the class LambdaExpr. To compile such a declaration, MultiJava creates a new class that contains the implementation in a static method. In this case, it generates something like:

```
class inferType {
    static Type inferType(LambdaExpr this_,
                                  TypeEnv env) { ... }
}
```

An important consequence of this compilation strategy is that it cannot support the addition of instance fields or abstract methods. This is an important restriction, since many extensions require both new fields and new methods. Furthermore, as we have seen, abstract methods are important for preserving the type-safe addition of new variants.

To use MultiJava's multiple dispatch mechanism, programmers annotate method declarations with argument *specializers*. For example:

```
class Rational extends Number {
    ...
    Number add(Number@Rational r) { ... }
    Number add(Number@Integer i) { ... }
    Number add(Number n) { ... }
}
```

This class declaration contains three implementations of the *add* method: one default implementation and two that provide specialized behavior for the cases when the argument is either a *Rational* or an *Integer*. When the application invokes *add* on a *Rational*, the system automatically performs a dynamic type test and selects the most appropriate implementation.

MultiJava compiles a set of multimethod definitions into a single method. In this case, for example, it translates the *Rational* class into something like:

```
class Rational extends Number {
    ...
    Number add(Number n) {
        if (n instanceof Rational) {
            Rational r = (Rational) n;
            ...
        } else if (n instanceof Integer) {
            Integer i = (Integer) n;
            ...
        } else {
            ...
        }
    }
```

}

There are two important consequences of this compilation strategy. One is that we cannot modularly extend this method to handle cases involving arbitrary new variants. The other is that, to preserve type safety, there must always be a default implementation of a method with abstract arguments. For example, if the *Number* class is abstract, then either every concrete subclass must provide a default implementation of *add*(*Number*), or the *add* method in class *Number* must not be abstract. Unfortunately, it is not always possible to provide meaningful defaults for such methods.

3 Unrestricted Open Classes and Multimethods

In this section, we describe our strategy for adding open classes and multiple dispatch to Java. Unlike MultiJava's, our approach permits arbitrary extensions and multimethods without sacrificing modular compilation.

Language Extensions

Our syntactic extensions are similar to those of MultiJava. Specifically, they include new constructs for open classes and multiple dispatch.

Our open class declarations are somewhat different from MultiJava's. Instead of defining extensions at the level of a single method, a programmer writes a compound `extensionof` declaration, whose syntax is as follows:

```
extensionof ⟨class⟩ {
    ⟨field | method declaration⟩*
}
```

This allows the programmer to bundle a set of related fields and methods into a single declaration, which is useful when there are dependencies between them.

For example, in the programming environment example from above, we might extend the *Expression* class with an abstract pretty-printing operation as follows:

```
extensionof Expression {
    static int tabWidth = 8;

    abstract void prettyPrint(PrintWriter out,
                              int indent);
}
```

Compiling this declaration extends the class with the new abstract method, as well as a new static field that the programmer can modify to customize the method's behavior. While this example adds a static field, our open class mechanism also permits the inclusion of instance fields.

To support multimethods, we allow argument specializers on formal parameters. For example, consider a hierarchy of numeric classes rooted at the abstract class *Number*. The class supports various binary operations over numbers, such as addition:

```
abstract class Number {
    abstract Number add(Number n);
    ...
}
```

In a concrete subclass representing integers, a programmer may write an efficient, specialized implementation of the *add* method:

```
class Integer extends Number {
    Number add(Number@Integer i) { ... }
}
```

Unlike MultiJava, our system does not require default methods. In this case, for instance, if *Integer* is the only concrete subclass of *Number* in the system, then the above declaration is sufficient. However, a programmer may decide to extend the system with a new class, for example *Rational*:

```
class Rational extends Num {
    Num add(Num@Rational r) { ... }
}
```

Attempting to compile this class fails for two reasons. First, because of the existence of the *Integer* class, *Rational*'s implementation of the *add* method is incomplete, so the programmer must at least add a method to handle *Integer*s. (If possible, he may also (or instead) add a default implementation.) In addition, the introduction of the *Rational* class into the system makes *Integer*'s implementation of the *add* method incomplete. The programmer must provide an extension of *Integer* containing the appropriate method or methods. Fortunately, in our system, the programmer can do this modularly (without access to *Integer*'s source code) through the open class mechanism. For example:

```
extensionof Integer {
    Number add(Number@Rational r) { ... }
}
```

4 Compilation Strategy

The ability of our language to express the sorts of modularly extensible programs described above depends heavily on our compilation strategy. In this section, we describe this compilation strategy in detail.

High-Level Approach

We aim to reuse existing compiler technology as much as possible. In particular, our compilation strategy does not require a custom code generator. Instead, our approach is to translate programs written in the extended language into ordinary Java, which we compile with a standard compiler. In addition, we extract semantic information from the original source code and, after compilation, apply bytecode transformations on the resulting classfiles to implement the extensions.

Compiling Open Class Extensions
To compile a class extension, we first translate it into a legal Java class declaration. Specifically, we generate source code defining a subclass of the class we intend to extend. For example, taking the example from above, we would translate

```
extensionof Expression {
    static int tabWidth = 8;

    abstract void prettyPrint(PrintWriter out,
                              int indent);
}
```

to

```
abstract class Expression_###
                extends Expression {
    static int tabWidth = 8;

    abstract void prettyPrint(PrintWriter out,
                              int indent);
}
```

and apply a standard Java compiler to this code. If compilation succeeds, we next *merge* all of the members defined in the resulting classfile into the *Expression* class. Since this merging procedure is critical to our compilation strategy, we take a brief digression to describe it in some detail.

Merging Classes
Java classfiles comform to a well-defined structure and contain a significant amount of semantic information aside from the virtual machine instructions of the methods they contain. Specifically, every classfile contains a *constant pool* that identifies the names and types of all the members declared in the class, as well as any external members on which the class depends.

Knowing the structure of a classfile makes it possible to copy all of the members of a subclass into its superclass. The procedure is as follows: first, internally change all occurrences of the subclass's name within its classfile to the superclass's name; next, import all entries from the subclass's constant pool into the superclass's constant pool; finally, copy all members of the subclass into the superclass, changing any references to the old constant pool to refer to the new (superclass's) constant pool.

While this simple procedure works for most cases, "special" methods, including *super* calls and constructors, require special treatment. In particular, for our purposes, we require that the subclass provide exactly the same set of constructors as the superclass, and that all constructor implementations simply invoke *super* with the same arguments. Then, when merging the classes, we can simply ignore the subclass's constructors. This allows us to avoid a number of issues, and we can easily enforce the requirement during our translation stage.

Another complication is that, if any method in the subclass overrides a method in the superclass but still makes use of the old version (through a *super* call), the merging process becomes more intricate. Specifically, the new method needs to override (in this case, replace) the old version, but the old version still needs to be available. The solution is to rename the old method to avoid conflicts, add the new one, and replace the new method's *super* call with a normal invocation of the (now renamed) method. We can perform this procedure through mechanical bytecode manipulations.

Compiling Multimethods
Our compilation strategy for multimethods involves generating dispatcher methods for each argument class of each multimethod. For example, imagine that we want to compile the *Number* example from above. The source code looks like:

```
class Rational extends Number {
    ...
    Number add(Number@Rational r) { ... }
    Number add(Number@Integer i) { ... }
    Number add(Number n) { ... }
}
```

Instead of translating the set of multimethods as a single method with dynamic type tests, we translate it into three implementation methods and generate dispatchers for the appropriate classes. The result looks like:

```
extensionof Number {
    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }
}

extensionof Integer {
    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }
}

class Rational extends Number {
    ...
    Number add_impl(Rational r) { ... }
    Number add_impl(Integer i) { ... }
    Number add_impl(Number n) { ... }

    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }

    Number add(Number n) {
        n.add_disp1(this);
    }
}
```

Note in particular that, in the resulting code, the method specializers have been removed. The implementation methods (the ones whose names end in *_impl*) correspond exactly to the original multimethod definitions. The *add* method itself, however, now simply calls a dispatcher, which in turn calls an implementation method according to the argument's dynamic type.

An important property of this compilation strategy is that it supports extensibility. For example, if a programmer later adds a new *Number* variant to our system, say *Complex*, he can modularly extend the system with a specialized method for adding *Rationals* to *Complex* numbers. He simply writes:

```
extensionof Rational {
    Number add(Number@Complex c) { ... }
}
```

which effectively compiles to:

```
extensionof Complex {
    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }
}
extensionof Rational {
    Number add_impl(Complex c) { ... }
}
```

In this case, the resulting code integrates naturally with the existing dispatching framework.

Things are slightly more complicated, however, if we want to compile an extension that specializes a preexisting singly-dispatched method. In this situation, we must first build the multiple-dispatching framework from scratch. Aside from the normal dispatcher generation, we need to rename the existing default method from [*meth*] to [*meth*]*_impl* and create a fresh version of [*meth*] that invokes the appropriate dispatcher.

The situation is most complicated when we specialize an abstract method without providing a default. In this case, the method must also have abstract arguments, and the system should only require that an implementation exist for each tuple of concrete arguments. As before, when compiling the implementation methods, we need to rename them and generate dispatchers for the argument classes. In addition to generating dispatchers for the concrete argument classes, we also need to add *abstract* dispatchers in the abstract superclasses of the arguments. This extra step enforces the constraint that an implementation must exist for each concrete subclass that we compile.

Since this mechanism is somewhat subtle, let us consider an example—specifically, the *Number* example from above. First, suppose we have the following declaration:

```
class Integer extends Number {
    Number add(Number@Integer i) { ... }
}
```

This effectively translates to:

```
class Integer extends Number {
    Number add_impl(Integer i) { ... }

    Number add_disp1(Integer arg0) {
        arg0.add_impl(this);
    }

    Number add(Number n) {
        n.add_disp1(this);
    }
}

extensionof Number {
    abstract Number add_disp1(Integer i);
}
```

In order for the *add* method in *Integer* to type-check, we need for *Number* to provide an *add_disp1*(*Integer*) dispatcher. Since we cannot provide a concrete implementation of this method in general, we add it abstractly. Note that *Integer* provides an implementation of this method, reflecting the fact that there is a method for adding two *Integers*. Furthermore, the presence of this abstract dispatcher method reflects the constraint that, if the programmer tries to add any concrete *Number* classes to the system, he must extend *Integer* with an implementation of *add* that handles the new variant. For example, an attempt to compile the following definition of a *Rational* class can only complete when accompanied by an appropriate extension of *Integer*:

```
class Rational extends Number {
    Number add(Number@Rational r) { ... }
    Number add(Num@berInteger i) { ... }
}

// required in this instance
extensionof Integer {
    Number add(Number@Rational r) { ... }
}
```

This translates to:

```
class Rational extends Number {
    Number add_impl(Rational r) { ... }
    Number add_impl(Integer i) { ... }

    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }

    Number add(Number n) {
        n.add_disp1(this);
    }
}

extensionof Number {
    abstract Number add_disp1(Rational r);
}

// this extension comes from the definition of Rational
extensionof Integer {
    Number add_disp1(Rational arg0) {
        arg0.add_impl(this);
    }
}

// this extension corresponds to the actual
// Integer extension
extensionof Integer {
    Number add_impl(Rational r) { ... }
}

// this extension comes from the extension to Integer
extensionof Rational {
    Number add_disp1(Integer arg0) {
        arg0.add_impl(this);
    }
}
```

Alternatively, if the programmer determines that it is possible to write generic *add* methods, he can compile these as extensions, which will replace the abstract dispatchers with concrete ones, reflecting the fact that the corresponding classes now require no additional implementations as new classes enter the system.

One problem we encounter when compiling multimethods for abstract generic functions is that the sorts of extensions that arise may be mutually dependent. Specifically, in the above example, we cannot compile the *Rational* class without having an implementation of *add*(*Rational*) in the *Integer* class. Conversely, we cannot compile the *add*(*Rational*) method until *Rational* exists. Standard Java compilers are designed to handle these sorts of situations in normal Java programs, but in our case the dependencies may involve extensions that can only be merged into their superclasses after compilation finishes. Here our translation strategy causes problems, since the Java compiler sees extensions as distinct subclasses.

To work around this problem, we can do one of two things. When the compilation process is stuck because of a dependency on a dispatcher, we can simply generate this dispatcher and merge it into the appropriate class. This operation is trivial because dispatchers have no dependence on the code for the methods for which they dispatch. The only problem is that, if compilation subsequently fails, we must be sure to remove the dispatcher. Alternatively, if the Java compiler requires the existence of a certain class before it can compile a particular extension, we can generate a *stub* for this class. In this case, the stub would contain all the members of the actual class, except that the method bodies would simply throw exceptions. Again, generating these stubs is easy, since they do not require code generation by an actual Java compiler.

Type-Checking

Because of the nature of our compilation strategy, our extensions do not introduce any new type-checking problems. That is, we provide a translation from our enriched language to standard Java which preserves type safety in a way that Java's ordinary, modular type-checking can detect. In particular, we have seen that our treatment of abstract class extensions and of multimethods with abstract top methods expresses completeness constraints directly in Java's type system.

Readers may be bothered by the fact that our open class mechanism allows programmers to extend abstract classes with new abstract methods, an action that effectively makes all existing concrete subclasses become abstract (or lose type safety, depending upon one's viewpoint). However, Java already gives programmers this capability—our system only extends the capability to situations in which source code is unavailable. In any case, whenever such a change occurs, all dependent classes, including subclasses, must undergo type-checking again. Thus, to build a perfectly "safe" system in the presence of such capabilities, a programmer needs a tool that tracks dependencies between classes and directs recompilation as necessary.

In addition to the problem of completeness, multimethod systems may suffer from ambiguous method invocations. These arise when more than one method applies to a given argument tuple but none is strictly more specific than all the others.

For example, consider a variation on the *Number* hierarchy from above:

```
abstract class Number {
    Number add(Number n) { ... }
    Number add(Number@Integer i) { ... }
}

class Integer extends Number {
    Number add(Number n) { ... }
}
```

This program contains three *add* methods—a default method, a partially specialized method for adding *Integers* to generic *Numbers*, and a partially specialized method for adding generic *Numbers* to *Integers*. If the system needs to add two *Integers*, all three of these methods apply. However, the system can eliminate the default method from consideration, since it is less specific than either of the specialized methods. Unfortunately, in trying to decide between the other two methods, there is no clear winner—each method is more specific in one argument position but less specific in the other.

It is always possible to resolve such ambiguities automatically, for example by *linearizing* the method-selection procedure, so that earlier arguments count more heavily than later ones. In the above scenario, for example, we might argue that the receiver is more important than the argument, in which case we should choose the method in the class *Integer*. However, there are reasons why this sort of resolution may be undesirable. In particular, an ambiguity may indicate an oversight by the programmer, in which case the type-checker should ideally raise a flag instead of silently assigning an arbitrary meaning to an underspecified system.

We call a system in which all arguments count equally *symmetric*. When such a system has no ambiguities, its behavior is not only robust but also highly predictable. In a symmetric system, the only way to resolve an ambiguity is to write a new method that is as specific as each of the ambiguous methods in every argument position. In the above scenario, for example, we would need to add a new method in which both the receiver and the argument are *Integers*.

In our system, dispatching proceeds from left to right through the argument list. Thus, it processes earlier arguments first. However, *this does not mean that our system performs linear dispatching*. On the contrary, our method-selection algorithm is symmetric, as we now explain.

When building the dispatching framework for a method, our compiler constructs a "net" of dispatchers that spans the cartesian product of argument specializers for all argument positions. As a result, after running through the full spectrum of dispatchers, the system knows enough about each argument's type to perform method selection. Furthermore, each path through the dispatching framework manifests itself statically as a final dispatcher in the chain—a dispatcher that calls an implementation method. In the above example, the compiler generates the following set of final dispatchers:

```
abstract class Number {
    Number add_disp1(Number arg1) {
        arg1.add_impl(this);
    }
    Number add_disp1(Integer arg1) {
        arg1.add_impl(this);
    }
}

class Integer extends Number {
    Number add_disp1(Number arg1) {
        arg1.add_impl(this);
    }
    Number add_disp1(Integer arg1) {
        arg1.add_impl(this);
    }
}
```

Essentially, we can view our dispatching framework as providing a bridge for moving dynamic types into the static type system. That is, it reduces the problem of dynamic method selection to that of static method selection. Importantly, Java's static overloading mechanism uses a symmetric lookup algorithm. Hence, our system also gets symmetry "for free" by virtue of its translation scheme. In the above scenario, for instance, Java's method-selection algorithm detects an ambiguity when trying to select which *add_impl* method to call from the last dispatcher.

5 Conclusions and Future Work

We have presented an extension to the Java language that supports open classes and multimethods. Unlike previous approaches, our compilation strategy permits unrestricted use of these features without resorting to global checking or compilation. Specifically, our strategy translates the extended language into standard Java in such a way that we can reuse ordinary Java type-checking and compilation almost exclusively. Importantly, our translation naturally expresses the completeness and unambiguity constraints of open-class/multimethod programs directly within Java's type system. We are currently in the process of developing an implementation of the system. Specifically, we have written and tested various pieces of it but have yet to integrate them all into a single, coherent tool.

Once the implementation is complete, we look forward to using it in our software development tasks. For example, various aspects of the compiler itself lend themselves to more natural expression through open classes and multimethods. Rewriting the compiler in our extended language would be an interesting experiment and would help both to verify its utility and to mold it into a useful tool. Other linguistic projects, including modular interpreters, compilers, and type systems, also benefit from the capabilities our language provides.

With use, we expect to find features that are not entirely satisfying. For instance, the merging technique that we use to implement open classes *mutates* the existing classes. As a result, it is not clear how one can easily undo the effect of compiling a particular extension. This property is contrary to the traditional models of modular software development, where each compilation unit corresponds to exactly one object module, and these modules can easily be compiled, re-compiled, or un-compiled. In the future, it might be worth investigating extending the system with an explicit linking process, so that class fragments can be compiled independently and later linked into their containing classes.

REFERENCES

[1] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification. *ACM SIGPLAN Notices*, 23(special edition):1–143, 1988.

[2] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[3] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 35(10), pages 130–145, 2000.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.

[5] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, June 1999. Springer Verlag.

[6] G. L. Steele, Jr. Building interpreters by composing monads. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, January 1994.