Half & Half: Multiple Dispatch and Retroactive Abstraction for Java™[†]

Gerald Baumgartner[*] Martin Jansche[**] Konstantin Läufer[***]

[*] Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
gb@cis.ohio-state.edu

[**] Dept. of Linguistics
The Ohio State University
Columbus, OH 43210
jansche.1@osu.edu

[***] Dept. of Computer Science
Loyola University of Chicago
Chicago, IL 60626
laufer@cs.luc.edu

March 23, 2002

Technical Report OSU-CISRC-5/01-TR08, Revised 3/02

Abstract

Software often goes through a variety of extensions during its lifetime: adding new fields or new variants to a data structure, retroactively creating new type abstractions, and adding new operations on a data structure. As characterized by the extensibility problem, it should be possible to apply any combination of these types of extensions in any order. Mainstream object-oriented languages, however, do not well support the latter two. This paper proposes two language mechanisms that facilitate extending existing type hierarchies: multimethod dispatch and retroactive abstraction. For these two mechanisms to coexist, it is necessary to allow method dispatch on parameters of interface types, which presents problems with static type-checking. We present a type-safe solution that combines the two mechanisms by limiting multimethod type checks to package boundaries and by compiling certain packages with multimethods into sealed Jar files.

1 Introduction

As software evolves, data structures might have to be extended along several dimensions. The nature of these extensions often cannot be foreseen at the design stage. Without appropriate support of the programming language and in the design of the software, simple extensions might require large amounts of existing code to be modified.

Suppose we are writing a compiler for a language the size of Java, where the data structure for representing the parse trees consists of 100 variants. If the compiler is written in an object-oriented style, we would represent this data structure as a class hierarchy with 100 classes. If later changes require adding additional information to certain parse tree nodes or adding new variants, the object-oriented style would make this straightforward by adding fields and

[†]Java is a registered trademark of Sun Microsystems, Inc.

methods to individual classes or by adding new classes to the hierarchy. If later changes require adding a new traversal of the parse tree, with the direct object-oriented style we would need to add one method to each one of the 100 classes. Similarly, adding new abstractions, such as new interfaces, would require changes to existing classes. Conversely, if the compiler were written in a functional style, adding new fields and adding new variants would be difficult, while adding new traversals would be straightforward.

Ideally, all of these extensions should be possible in any order without requiring existing code to be modified.

Single dispatch is used in traditional object-oriented languages to select and invoke a method based on the run-time class of the single, distinguished receiver argument of the method invocation. This mechanism allows adding new classes to an existing hierarchy or overriding methods in subclasses. Single dispatch also facilitates encapsulation of data within a class.

By contrast, *multiple dispatch* is a mechanism for dispatching a method at run time based on the classes of any subset of the arguments. While single dispatch is the appropriate mechanism for the object-oriented programming style of defining data types as extensions of existing classes, multimethods provide better support for an abstract data type or functional programming style. Also, multimethods provide a semantically simpler alternative to static overloading.

Multimethods facilitate the development of object-oriented software in various ways. Notably, they allow clients of existing classes to add new operations that dynamically dispatch on arguments of these classes. Multimethods thus support a notion of *open objects* [29] of adding new operations to existing data structures. Multimethods also support safe covariant overriding [8] and binary methods [7].

For example, suppose we need to add a new tree traversal operation, such as a type checker, to our parse tree data structure. Instead of the traditional object-oriented style of adding one method to each one of the 100 classes, which risks breaking existing code and distributes the control flow of the type checker over many source files, multimethods allow adding a single type checker class that distinguishes up to 100 individual cases (one per class in the hierarchy). Without support for multimethods, the Visitor pattern [19] could be used for achieving a similar program structure. The Visitor pattern, however, requires the programmer to write the dispatch mechanism by hand and does not support adding new classes to the hierarchy or adding traversal methods with additional or different parameters.

Most object-oriented languages provide some form of *subtype polymorphism*, allowing an instance of a type to be used wherever an instance of a supertype of that type is expected. Usually, a subtype relationship is defined *by name*. The subtype explicitly names its supertype(s) in the type declaration. By contrast, with *structural subtyping*, any type (class or interface) that implements each method of an interface is a subtype of the interface, and any expression of a structural subtype can be used wherever a value of the structural supertype is expected. Structural subtyping makes the type system more flexible in situations that require *retroactive abstraction* over existing types [25].

For example, suppose we are given a precompiled library class `RandomDataAccess`. We want to abstract over this class with the restricted interface `ReadOnlyRandomAccess` that does not include the `write()` method and then add class `CDRomDrive` as an alternative implementation. With named subtyping, we either need to modify the existing class by declaring that it `implements` the new interface or write glue code in the form of the Adapter pattern [19]. Structural subtyping allows abstracting over existing code by simply adding an appropriate interface declaration.

Similarly, structural subtyping can be used for retroactive abstraction over a subset of the classes of an existing hierarchy such that a new traversal can be added to work on only that subset of classes. The new traversal would then be implemented using multimethods. Since the retroactive abstractions are interfaces, this requires the multimethod dispatch to be performed on a parameter of an interface type, which causes problems with static type-checking. A similar need for multimethods that can dispatch on interface types arises when adding support for closures and function types to the language [2].

This paper explores the implications of combining encapsulated multimethods and retroactive abstraction. Millstein and Chambers [29] have described a spectrum of restrictions for multimethods with trade-offs between expressibility and modularity. By adding encapsulated multimethods and retroactive abstraction to Java, we achieve modularity using classes for encapsulation. We allow interfaces as argument types of multimethods at the expense of additional compile-time type checks through regional program analysis. Instead of full structural subtyping, we provide a retroactive abstraction declaration for abstracting over existing classes or interfaces. Parasitic methods [6] also allow interfaces as argument types but have a potential ambiguity problem. MultiJava [13, 14], on the other hand, does not allow interfaces as multimethod argument types. Our extension of Java with multimethods and structural subtyping is a step in the development of Brew, a successor language for Java with a new object model [3]. We are currently implementing Half & Half, our extension of Java with multimethods and retroactive abstraction, as a front end of the Brew compiler.

2 Motivation

Writing extensible software requires different ways of extending existing type hierarchies: adding variants to a data structure, retroactively creating new type abstractions, and adding new operations on a data structure. Such extensions are not well supported in existing object-oriented languages; the *extensibility problem* [31, 15, 24, 36, 18] characterizes the current difficulties in addressing these needs.

2.1 The Extensibility Problem

The extensibility problem considers typical paths along which software systems evolve and, following Findler and Flatt's description [18], can be summarized as follows. A software component might consist of a data type in the form of different variants, along with operations on those variants. A client of such a component may want to use the component in three different ways:

- The client may want to use the component in its original form.
- The client may want to *add a variant* to the data type; this usually requires adding a case for the new variant to each existing operation.
- The client may want to *add an operation* to the component; this usually requires including in the new operation a case for each existing variant.

To facilitate software maintenance and to accommodate components available only in compiled form, such uses must be possible without modifying or recompiling

- the original program component (data type and operations),
- existing client code, or
- existing components added as new variants to the original datatype.

The last constraint—*retroactive abstraction* to allow existing, possibly compiled, components to become variants of a data type—appears to be a novel aspect of the discussion of the extensibility problem.

In object-oriented programming, it is common to model a data type with variants through subtyping. A root type specifies the common operations available on the datatype, and its subtypes represent the variants. We use the following example, based on work by Krishnamurthi, Felleisen, and Friedman [24], to compare different object-oriented attempts at solving the extensibility problem. Specifically, the example involves a data type with root type `SimpleShape` and two original variants, `Circle` and `Trans`(lation), as well as some original operations on shapes. Another variant as well as other operations will be added later. The common declarations for the data type are given here:

```
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Shape;

interface SimpleShape {
    java.awt.Rectangle getBounds();
    // other original shape methods
}

class Circle implements SimpleShape {
    double r;
    public Circle(Point p, double r) {...}
    public Rectangle getBounds() {...}
    // other original shape method
    // implementations for Circle
}
```

```
class Trans implements SimpleShape {
    Point p;
    SimpleShape s;
    public Trans(Point p, Shape s) {...}
    public Rectangle getBounds() {...}
    // original shape method
    // implementations for Trans
}
```

2.2 Extensibility with Encapsulated Multimethods and Retroactive Abstraction

Based on the aforementioned example, we will now illustrate the combination of retroactive abstraction and encapsulated multimethods.

The interface `java.awt.Shape` from the Java Abstract Window Toolkit [35] is an existing component that we wish to add as a variant; we use retroactive abstraction to declare `java.awt.Shape` as a subtype of `SimpleShape`.

```
interface java.awt.Shape extends SimpleShape;
```

To add a new operation to the component, we provide the class `Containment`, which defines the multimethod `contains`. This multimethod consists of a *generic method* that provides the static type signature of the multimethod, as well as a case for each existing variant of the data type.

```
class Containment {
    public generic boolean contains(SimpleShape s, Point o);

    public boolean contains(SimpleShape s, Point o) {...}
    public boolean contains(Shape s, Point o) {...}
    public boolean contains(Rectangle r, Point o) {...}
    public boolean contains(Circle c, Point o) {...}
    public boolean contains(Trans t, Point o) {
        return contains(t.s, new Point(o.x - t.p.x, o.y - t.p.y));
    }
}
```

To add a new variant to the data type, we define the class `Union`. For using the operation `contains` on the new variant, we provide the class `UnionContainment`, an extension of `Containment` that adds a case to the multimethod `contains` to handle the new variant.

```
class Union implements SimpleShape {
    SimpleShape s, t;
    public Union(SimpleShape s, SimpleShape t) {...}
    public Rectangle getBounds() {...}
}

class UnionContainment extends Containment {
    public boolean contains(Union u, Point o) {
        return contains(u.s, o) || contains(u.t, o);
    }
}
```

The following example of client code illustrates the application of the multimethod to a shape and a point. As operations are added in the form of multimethods, clients can continue to use the original variants to construct instances of the data type.

```
new UnionContainment().contains(
    new Trans(new Point(3, 4), new Union(new Circle(3), new Circle(5))),
    new Point(2, 3))
```

2.3 Existing Approaches to Extensibility

We will now take a look at existing object-oriented approaches to solving the extensibility problem.

It is easy to add new variants to the original example in the form of new subtypes, such as `Union`.

```
class Union implements SimpleShape {
    SimpleShape s, t;
    public Union(SimpleShape s, SimpleShape t) {...}
    // shape method implementations for Union
}
```

It is difficult, however, to add new operations to the data type, since this requires adding a method to the root type as well as each existing subtype. Furthermore, doing so requires modifying existing client code to use the extended variants instead of the original ones.

```
interface SimpleShape {
    Rectangle getBounds();
}

// original variants...

interface ShapeWithContainment extends SimpleShape {
    boolean contains(Point o);
}

class CircleWithContainment extends Circle implements ShapeWithContainment {
    public boolean contains(Point o) {...}
}

// extensions of the other variants...
```

The Visitor pattern [19] provides a partial solution to this problem. Here the root type specifies only a single dispatch operation that takes a visitor (an object representing an operation on the data type) as an argument. Based on the assumption that the data structure is stable, the visitor interface requires a method for each existing variant of the data type. The Visitor pattern makes it easy to add new operations in the form of a single class.

```
interface SimpleShape {
    Object accept(ShapeVisitor v);
}

interface ShapeVisitor {
    Object visitCircle(Circle c);
    Object visitTrans(Trans t);
}

interface UnionVisitor extends ShapeVisitor {
    Object visitUnion(Union u);
}

class Circle implements SimpleShape {
    // ...
    public Object accept(ShapeVisitor v) {
        return v.visitCircle(this);
    }
}

// implementations of the other variants...

class Containment implements ShapeVisitor {
    Point o;
    public Containment(Point o) { this.o = o; }
    public Object visitCircle(Circle c) {...}
    public Object visitTrans(Trans t) {...}
}
```

However, adding new variants is now hard, since it requires extending the visitor interface. Doing so requires modifying each existing visitor class to include a method for the new variant.

```
class Union implements SimpleShape {
    SimpleShape s, t;
    public Union(SimpleShape s, SimpleShape t) {...}
    public Object accept(ShapeVisitor v) {
        return ((UnionVisitor) v).visitUnion(this);
    }
}

class UnionContainment extends Containment implements UnionVisitor {
    public UnionContainment(Point o) {
        super(o);
    }
    public Object visitUnion(Union u) {...}
}
```

We observe an interesting duality between the two approaches. In both cases, it is necessary to plan ahead to support the particular programming protocol so that later modifications of existing code can be avoided.

- If ordinary subtyping is used to add operations to the data type, then client code should be parameterized ahead of time by an Abstract Factory [19]. In the absence of a decent module language [18], this parameterization plays the role of a simple dynamic linker and avoids hard-coding the client to a specific data type implementation.

```
interface ShapeFactory {
    SimpleShape createCircle(double r);
    SimpleShape createTrans(Point p, SimpleShape s);
    // ...
}

class SimpleShapeFactory implements ShapeFactory {
    public SimpleShape createCircle(double r) {
        return new Circle(r);
    }
    // ...
}

class ShapeFactoryWithContainment implements ShapeFactory {
    public SimpleShape createCircle(double r) {
        return new CircleWithContainment(r);
    }
    // ...
}

// client with ShapeFactory parameter

factory.createTrans(new Point(3, 4),
                    factory.createUnion(factory.createCircle(3),
                                        factory.createCircle(5))))
```

- If operations are implemented as visitors, then any visitors that create new visitor instances should implement and use Factory Methods [19] for this purpose [24], which are a special case of Abstract Factory where the factory's client also acts as the factory itself. The Visitor pattern limits the arguments of each visitor method to a single data type variant. The resulting stateful visitors may need to create additional visitor instances with modified state to model the partial application of the same function to a modified argument. Without this technique, a `Union` shape within a `Trans` shape would lead to a run-time exception at the cast down to `UnionVisitor` in the method `Union.accept`.

```
class Containment {
    // ...
    public Containment createContainment(Point o) {
        return new Containment(o);
    }
    public Object visitTrans(Trans t) {...}
        return t.s.accept(createContainment(new Point(o.x - t.p.x, o.y - t.p.y)));
    }
}

class UnionContainment extends Containment {
    // ...
```

```
            public Containment createContainment(Point o) {
                return new UnionContainment(o);
            }
            public Object visitUnion(Union u) {
                return new Boolean(((Boolean) u.s.accept(this)).booleanValue()
                                   || ((Boolean) u.t.accept(this)).booleanValue());
            }
        }
```

Since the Visitor pattern requires visitor support in each variant of the data type, it does not directly accommodate retroactive abstraction involving existing components. By contrast, using ordinary subtyping allows a limited form of retroactive abstraction by adding a variant that is a subtype of the existing component and a subtype of the data type's root type.[1]

Recently, Zenger and Odersky [37, 38] presented a new programming protocol for extensible visitors. This protocol is based on the Visitor pattern, making it easy to add operations, but overcomes the limitations of the Visitor pattern by adding default cases to operations; a default case handles future extension of the data type on which an operation is used. Since the protocol is complex, language support for extensible data types would be desirable.

3 Language Design

We argue that the combination of multimethods and retroactive abstraction has the advantages of the existing approaches while avoiding some of their drawbacks. Specifically, our approach has the following favorable characteristics:

- No support for the Visitor protocol is needed in the data type.
- No parameterization of the client code with an Abstract Factory or other form of dynamic linking protocol is needed.
- Operations on the data type can be added in the form of new multimethods as easily and elegantly as in the Visitor pattern.
- Variants of the data type can be added easily.
- Existing types can be added as variants of the data type using retroactive abstraction.
- Ordinary methods and multimethods can coexist in the datatype.
- Multimethods can be type-checked statically.
- More specific return types for operations can be defined. In contrast, the Visitor pattern supports only a single return type, typically `Object`.
- Additional arguments for operations can be defined. In contrast, the Visitor pattern requires such information to be part of the state of the visitor, requiring Factory Methods to support the extension of the visitor state.

In the remainder of this section, we present each of the two proposed language mechanisms, retroactive abstraction and multimethods, from a language design perspective.

3.1 Retroactive Abstraction

Consider an application that performs read-only random access on databases located either on disk or on a CD-ROM drive [25]. We access disk files using the class `java.io.RandomAccessFile`, but using only the methods for reading from the file, not the ones for writing to the file.

[1]This is no problem in Java as long as the root type of a data type is an interface.

```
class RandomAccessFile implements DataOutput, DataInput {
    public void close() throws IOException {...}
    public FileDescriptor getFD() throws IOException {...}
    public long getFilePointer() throws IOException {...}
    long length() throws IOException {...}
    void seek(long pos) throws IOException {...}
    // implementations of methods from the
    // interfaces DataOutput and DataInput
}
```

We also provide a class for accessing information on CD-ROM drives. Since CD-ROM drives are read-only devices, the corresponding class implements only interface `java.io.DataInput` and provides additional methods for random access:

```
class CDRomDrive implements DataInput {
    long length() throws IOException {...}
    public void seek(long pos) throws IOException {...}
    // implementation of methods
    // from interface DataInput
}
```

Besides the methods for reading data, the application needs to use the random-access method `seek`. We therefore would need an interface of the form

```
interface ReadOnlyRandomAccess extends DataInput {
    long length() throws IOException;
    void seek(long pos) throws IOException;
}
```

Unfortunately, the Java library does not provide such an interface; The methods `length` and `seek` are not provided in either of the interfaces `DataInput` and `DataOutput`.

Since class `RandomAccessFile` is part of the library and cannot be modified, we would have to write an Adapter class [19] that implements the interface `ReadOnlyRandomAccess` by name and forwards requests to class `RandomAccessFile`.

Instead of requiring users to write and use such an Adapter class, we provide a new language construct that retroactively establishes a subtype relationship between an existing class and an interface.

```
class RandomAccessFile implements ReadOnlyRandomAccess;
```

Such a retroactive abstraction declaration can appear in any context in which regular class declarations are allowed and where both the existing class and the interface are visible.

As long as class `RandomAccessFile` structurally conforms to interface `ReadOnlyRandomAccess`, we allow the introduction of such retroactive subtype relationships. The type check performed for a retroactive abstraction is the same as if the `implements` clause were added to the original class definition.

It is the programmer's responsibility to make sure that an interface that retroactively abstracts over an existing class has a specification that is met by the implementation provided by that class, just as it is the programmer's responsibility in standard Java (and many other object-oriented languages) to ensure that a class that implements an existing interface obeys the specification of that interface.

Similarly, for retroactively abstracting over an existing interface `J` with a new interface `I`, we provide the construct

```
interface J extends I;
```

Both of these retroactive abstraction relationships could be inferred from the code instead of declared [25, 5, 4, 1]. E.g., given an assignment of the form

```
ReadOnlyRandomAccess f = new RandomAccessFile();
```

the compiler could verify that the right-hand-side class implements all the methods of the left-hand-side interface, i.e., that the class is a structural subtype of the interface. However, this has the disadvantage that accidental retroactive abstractions where the class does not satisfy the intended semantics of the interface are difficult to track down. A retroactive `implements` declaration is more visible in the code and can be regarded as an abbreviation for an Adapter class.

3.2 Multimethods

Multimethod languages without single dispatch, such as Common Lisp [30, 33], Dylan [32, 17], or Cecil [10, 12, 9], provide symmetric multimethods, where the dynamic dispatch is performed on all arguments symmetrically. MultiJava [13, 14] adopted this style of multimethods as well.

For an extension of a single dispatch language such as Java, we argue that purely symmetric multimethods are undesirable since they do not fit into the language as well as encapsulated multimethods. Java allows the receiver of a method to access the private fields of the surrounding class. With symmetric multimethods, a method would either have privileged access to the classes of all parameters or to none. While the former would violate Java's encapsulation, the latter is often too restrictive. If only one parameter is treated differently for encapsulation purposes, then this should be the receiver as for other methods in Java.

Java already has overloaded methods, where out of a set of methods with the same name the appropriate method signature is selected at compile time based on the static types of the arguments. Multimethods are similar to overloaded methods, except that method selection happens at run time. Removing overloading from the language is not an option since it would break existing code. For the two mechanisms to coexist, we argue that it is desirable that multimethods can be used exactly the same way and in the same contexts as overloaded methods with the only exception being that method selection happens at run time.

By defining multimethods, like overloaded methods, as methods inside a class and by using the same semantics for the receiver argument, multimethods become encapsulated. Encapsulated multimethods are useful for defining binary methods [8, 7, 6] but they also provide additional flexibility for writing visitors. Suppose we want to add two `draw` operations to our `SimpleShape` hierarchy for drawing shapes in different styles. By writing these operations as encapsulated multimethods and by using an abstract factory, we can parameterize a client of `draw` to use either of the two `draw` implementation. If this additional parameterization is not required, a visitor can also be defined as a set of symmetric multimethods by declaring them static.

Like a set of overloaded methods in Java, we define a *generic function* as a set of methods (its *multimethod* components) with the same name but with different signatures. Unlike an overloaded method, which is selected at compile time based on the static type of the arguments, a multimethod component of a generic function is selected at run time based on the dynamic types of the arguments.

We make the signature of the generic function explicit by requiring the programmer to declare the generic function by providing a method signature modified by the keyword `generic`. In the following example, the generic function `print` is implemented by two multimethods:

```
public generic void print(C x);
public void print(C x) { ... }
public void print(D x) { ... }
```

where type `D` must a subtype of type `C`. The types `C` and `D` can be interface types as well as class types. We do not allow dispatch on basic types since Java does not define subtype relationships for them.

For dispatching on null pointers, we also provide the syntax

```
public void print(null) { ... }
```

using the pattern `null` instead of a parameter declaration. If no multimethod is defined for dispatching on null, the multimethod for the most general parameter type is selected. It is an error if there is no unique method with a most general parameter type.

In order to avoid any confusion between multimethods (dynamic overloading) and static overloading we do not allow multimethods to be mixed with static overloading. In the above example, the declaration of the generic function forces all other methods with the same name to be treated as multimethods of that generic function. It is a compile-time error if the class where the generic function is first declared inherits any methods with the same name as the generic function.

The ban on mixing dynamic and static overloading is not strictly necessary. It is conceivable to have a generic function and another method with the same name that is not a multimethod of the generic function (e.g., we could label multimethods with a special keyword to distinguish them from methods that statically overload the generic function). However, this would complicate method selection tremendously and has few, if any, practical uses. The only reason for allowing this would be to work around Java's distinction between simple types (`int`, `double`, etc.) and reference types. It might make sense to allow static overloading of a generic function with methods that have one or more parameters belonging to a simple type.

The generic function declaration may be modified by access modifiers, and may be declared `static` and/or `final` (and in addition `synchronized` and/or `strictfp`, but these merit no further discussion). It may, however, not be declared `abstract`. If the generic function is final, this means that the set of its multimethods may not be extended in a subclass, although individual multimethods may still be overridden. If the generic function is static, this means that no dynamic dispatch based on the receiver argument takes place, only symmetric dispatch based on all argument types, i.e., we are dealing with ordinary symmetric multimethods.

Multimethods must be declared to have the same protection as the generic function they are part of. They can be further modified by `abstract` or `final` (and/or `synchronized` and `strictfp`). If one or more multimethods are abstract, the surrounding class must be abstract. If a multimethod is final, this means, as usual, that it cannot be overridden in a subclass. Like regular methods, non-final multimethods can be overridden in a subclass. If a multimethod calls `super` the overridden multimethod from the super class is executed.

If a multimethod declares that it throws an exception, the generic function must declare a supertype of that exception. Likewise, the result type and each parameter type of the generic function must be a supertype of the respective result type or parameter type of each multimethod.

To resolve a multimethod call of the form `x.print(y)`, we first look for a declaration of a generic function `print` in the static type of `x`, where the type of `y` must be a subtype of the parameter type of `print`. Then at run-time, the generic function selects the appropriate multimethod. If the multimethod was overridden in a subclass, Java's single dispatch mechanism then selects the appropriate method from the dynamic type of the receiver. These steps in selecting a method are the same as for overloading, except that in case of overloading the second step happens at run time.

4 Type-checking

4.1 Retroactive Abstraction

Retroactive abstraction requires only a few changes to Java's type-checking algorithm, affecting mainly assignments of values to (the equivalent of) L-expressions [25]. These occur in a small number of places, namely explicit casts, assignment statements, passing values to the parameters of a method on method invocation, and assigning method results using a `return` statements (also `throw` statements, but since the type of the thrown expression must be a class type, we can ignore this case here).

Type-checking proceeds by examining the declared type of the expression being assigned to and of the assigned expression. We first check whether the expression being assigned to is a supertype of the type of the assigned value, in which case the assignment statement is well-typed. Otherwise we carry out any other standard Java type-checks for assignment. The supertype relation is the reflexive transitive closure of the union of the `extends` and the `implements` relations, possibly declared retroactively.

Checking retroactive abstraction declarations of the form

```
class C implements I;
```

is straightforward: exactly the same type-check is carried out as for an ordinary `implements` clause of a class declaration.

Similarly, the type check required for checking

```
interface I extends J;
```

is exactly the same as for type-checking an ordinary `extends` clause of an interface declaration.

As explained below, the subtype relation defined by a retroactive abstraction declaration is needed for performing the multimethod completeness and uniqueness type checks. To allow these type checks to be performed across class boundaries, the subtype relation must be recorded in a class file. The byte code format [28] allows user-defined attributes to be included in a class file; standard virtual machines would simply ignore these attributes. We use this mechanism to record the subtype relation defined by a retroactive abstraction declaration in the byte code file for the class containing the declaration or, if the declaration is outside class scope, in the byte code file for the public class of the source file.

4.2 Multimethods

Generic functions and their multimethods require no special type-check on the client-side, i.e., where a generic function is invoked. The method invocation type-check, including the above modifications, is carried out on the basis of the declared signature of the generic function.

The generic function does, however, require an elaborate type check on the implementation side, i.e., when it is declared or extended with new multimethods. A generic function is declared exactly once, and we refer to its declaration site as the *base class of* the generic function. This means that when a declaration modified by the keyword `generic` is encountered, we must first check that it does not override a generic function from a supertype (hiding, however, is allowed for static generic functions). For each method not modified by `generic`, check if there exists a generic function with the same name and a conforming type, which can be declared in the same class or inherited from a superclass; if there is, the method is a multimethod of that generic function.

Next, the set of all multimethods of a generic function is determined. It consists of all multimethods (with the same name and a conforming type) declared in the current class and any multimethods (with the same name and a conforming type) available in the superclass, except those that are overridden by a multimethod in the current class. Overriding of multimethods is the same as method overriding in standard Java.

Dispatch based on the non-receiver arguments is symmetric and requires the usual checks for symmetric multimethods, namely:

completeness: for any argument tuple there must exist at least one applicable multimethod; and

uniqueness: for any argument tuple there must exist at most one most specific applicable multimethod.

Since we allow interface types as argument types of multimethods, the usual ways of carrying out these checks [29, 14] are not directly applicable. For example, consider the following declarations, where `I` is an interface, and `J` and `K` are sub-interfaces of `I` but neither is a sub-interface of the other:

```
generic void print(I);
void print(null) { ... }
void print(J x)  { ... }
void print(K x)  { ... }
```

If there is a class that implements both `J` and `K`, uniqueness is violated. Type-checking needs to ensure that such a class does not exist. However, the information needed to check this is not necessarily available at compile-time in the case of separate compilation when the full type hierarchy is not known.

To remedy this situation, we propose using a regional link-time check at the level of a Java package. More precisely, we impose the following restriction:

> For any two multimethods of the same generic function, if one has a formal parameter of an interface type, the other has in the same position a formal parameter of an interface type, and the two interface types are incomparable (neither is a subtype of the other), then at most one of the interfaces can be publicly visible or can be (retroactively) extended by a publicly visible interface.

In the previous example, if interface `J` is public but `K` is not and does not have a public subinterface (so `K` must be declared in the same package as the class containing the generic function), there can be no class *outside* the package of the generic function that implements both `J` and `K`, since `K` and any subinterfaces of `K` are not visible outside its package.

However, we still need to make sure that there is no class *inside* the package of the generic function that implements both `J` and `K`. This is problematic in standard Java, since packages are by default open in the sense that one may retroactively add classes to an existing package. For example, it is possible to create a file containing the following Java code:

```
package java.lang;
public class Foo {}
```

If this file is compiled and the resulting bytecode file is placed in the classpath of the Java virtual machine, one can then write code such as this:

```
class Bar { Foo f; }
```

An explicit `import` declaration is not needed, since the package `java.lang` is imported by default, hence the name `Foo` is visible. This is what we mean when we say that the `java.lang` package is open.

In ordinary Java, this lack of modularity is not a major problem. However, in the presence of multiple dispatch we are faced with a problem of missing encapsulation: in the example above we noted that a class that implements both `J` and `K` would result in a violation of the uniqueness property that allows unambiguous resolution of a multimethod invocation; but if a package is open nothing could stop a developer from adding such a class after the generic function in this example has been successfully compiled, potentially giving rise to a uniqueness violation that cannot be detected at compile time.

In a traditional compilation model in which linking is controlled by the developer, an appropriate type-check could be carried out at link-time when the entire class hierarchy is known (but see below on dynamic loading of classes). However, this approach does not fit into the standard Java compilation model. Instead, we propose the use of *package sealing*, which is already part of Java. A package can be marked as sealed in a Jar file, which means that all of its classes must be loaded from the same Jar file. If the compiler writes its bytecode output into Jar archives it can attain a certain amount of control over the class inventory of a package. In particular, if there are restrictions on what interfaces a class may implement because those interfaces are used for dispatching multimethods, this restriction can be recorded inside the Jar file for the package and the compiler can then ensure that all classes comply with any such restrictions and, if necessary, seal the package. This compilation scheme requires no changes to a JVM with class loaders that respect the integrity of sealed packages.

Notice also that this introduces a kind of *closed world assumption*, since package-local classes and interfaces can no longer be extended, but all their subtypes are known when the package is sealed. In other words, package-local types now behave like disjoint sum types in functional languages (e.g., in the ML family).

The only way whereby a class that implements both `J` and `K` could still be present at run-time is via explicit dynamic loading [27]. The compiler can generate a special class loader that carries out any necessary checks when a class is loaded at run time and throws a linkage error in case a problem is detected.

With the above restriction and the package-level check in place, the rest of the implementation-side checks proceed as usual [12]: construct a partially ordered set of tuples of the form $(t_1, \ldots, t_n)$ where each t_i is a (retroactive) subtype of the type of the ith parameter of the generic function and a (retroactive) supertype of the type of the ith parameter of some multimethod; then check that the completeness and uniqueness properties are satisfied for all elements of this set.

For enforcing multimethod completeness we use the same restriction as in MultiJava [13, 14, 29], but for package boundaries instead of file boundaries:

> If a generic method has an abstract class or an interface as a parameter type and a dispatch is performed on this parameter, then there must be a multimethod implementation with the same parameter type, unless the abstract class or interface is only package visible and does not have a public subclass or subinterface for which no multimethod is defined.

E.g., the generic function `print` above does not have a multimethod of type `print(I)`. If `I` is public, an error is reported saying that the multimethod is potentially incomplete because `I` could be implemented by a class outside the package. If `I` has only package visibility and there is no class within the package that only implements `I` but neither `J` nor `K`, then the generic function type-checks. In the latter case, the package must also be sealed in a Jar file to prevent a public subtype of `I` being added without re-checking the multimethod.

For constructing this partially ordered set of tuples, as well as for checking the restriction on public interfaces, the compiler must collect all retroactive abstraction relations from the package. These were recorded in the byte code files when the corresponding declarations were type-checked.

5 Compilation

Compilation is best conceptualized as a source code transformation that maps code containing retroactive abstraction declarations or multimethods into ordinary Java code. We will implement these language mechanisms in the Brew compiler as a transformation of the parse trees.

5.1 Retroactive Abstraction

In previous work we have described an extension of Java with structural subtyping [25]. The implementation approach of modifying the type checker in the virtual machine we described in that paper also works for retroactive abstraction. Because of the large installed base of Java VMs, it is preferable, though, for the compiler to generate standard byte code that can run on any virtual machine.

If modifying the virtual machine is not an option, retroactive abstraction can be implemented by using adapter objects that take the place of objects of classes that have been retroactively abstracted over. The basic idea is illustrated by the following example [25]:

```
interface I {
    byte aMethod();
    void anotherMethod(int x);
}

class C implements I;
```

Here, `C` is an existing class type for which source code may not be available. The retroactive abstraction declaration gives rise to the following adapter class. The name of the adapter class is not a legal Java identifier, but can nevertheless be represented in Java bytecode.

```
class Ad@pter_C_I implements I {
    final C instance;
    Ad@pter_C_I(C i) {
        instance = i;
    }
    public byte aMethod() {
        return instance.aMethod();
    }
    public void anotherMethod(int x) {
        instance.anotherMethod(x);
    }
    public int hashCode() {
        return instance.hashCode();
    }
    // etc. for other specified methods
}
```

In other words, an adapter object forwards all methods from the interface that `C` has been declared to implement, including those from `java.lang.Object`, to a `C` instance it contains.

Consider a code fragment such as the following:

```
class C implements I;
C c;
I i = c;
```

Now the last assignment statement can be translated into

```
I i = new Ad@pter_C_I(c);
```

This translation will be revised in the subsequent discussion and serves mostly as an illustration of the basic idea. Such a translation is possible because the adapter object assigned to `i` is actually an instance of interface `I` in ordinary Java.

A few problems remain, which arise from the fact that this compilation scheme uses potentially many different objects to represent a single `C` object, namely the object itself plus any number of adapter objects. For example, the following code snippet is problematic:

```
C c = Factory.createC();
I i = c;
if (c == i) // ...
```

In this case, the expression `(c==i)` should evaluate to `true` no matter at what point `C` was declared to implement `I`. But the assignment to `i` created an adapter object that is different from the value of `c`, hence the pointer comparison on the last line will fail.

The general remedy is to use adapter objects very cautiously and only in contexts where they are absolutely necessary. Note that Java's `==` operator on reference types can be thought of as having been declared as

```
boolean ==(Object x, Object y);
```

This means that in `(c==i)` both `c` and `i` are implicitly cast to `Object`. But in that case the adapter is no longer needed, since its sole purpose was to disguise `C` objects as `I` objects; for purposes of pointer comparison we always want to get rid of any adapters.

We must modify the translation scheme slightly so that the environment contains the following interface:

```
public interface Ad@pter {
    Object getInstance();
    Object toObject();
}
```

Now we define the following transformer of expressions that discards any adapters, if present:

```
unwrap(x) := ((x instanceof Ad@pter) ? ((Ad@pter)x).toObject() : x)
```

The above adapter has to be modified slightly to implement the `Ad@pter` interface:

```
class Ad@pter_C_I implements Ad@pter, I {
    final C instance;
    Ad@pter_C_I(C i) {
        instance = i;
    }
    public Object getInstance() {
        return instance;
    }
    public Object toObject() {
        return unwrap(instance);
    }
    // remainder unchanged
}
```

We can now translate a pointer comparison of reference types such as `(c==i)` into

```
unwrap(c) == unwrap(i)
```

This translation clearly preserves the intended semantics of pointer comparison in the presence of retroactive abstraction via adapters.

A similar issue arises with `instanceof` expressions. If we have an object `c` of class `C` that is a subtype of `I`, no matter where that subtype relationship was declared, then `(c instanceof I)` must evaluate to `true`. The following nonstandard cases arise when retroactive abstraction is involved: `c` is an adapter object and `I` is a supertype of the object wrapped up in `c`, but not of `c` itself. In this case `(unwrap(c) instanceof I)` evaluates to `true`. Or `c` is not an adapter object but a `C` object, and `I` was retroactively declared as a (not necessarily immediate) supertype of `C`. In that case we need a separate mechanism for keeping track of retroactively introduced subtype relations.

We need a global data structure that keeps track of the nonstandard subtype relations. The following design is meant to be compatible with on-demand class loading:

```
class Registry {
    // Register c as a subtype of i
    // and use a as the mediating adapter.
    public static void register(Class c, Class i, Class a) {...}

    // Test whether c is an instance of i.
    public static boolean conforms(Object c, Class i) {...}

    // Return a representation of c that is
    // an instance of i; throw an exception
    // if the conversion does not succeed.
    // Conversion is guaranteed to succeed
    // if conforms(c,i) is true.
    public static Object convert(Class i, Object c) throws RuntimeException {...}
}
```

Here the `conforms()` method is analogous to an `instanceof` test, and the `convert()` method mirrors a cast expression, as we shall see shortly.

Now we can translate an expression like `(c instanceof I)` into

```
Registry.conforms(c, I.class)
```

The registry also handles all type conversions of class or interface types, including explicit casts, but also implicit upwards casts in assignment statements, parameter instantiations, `return` statements, etc. Explicit casts, such as

```
(I) c
```

are translated as

```
(I) Registry.convert(I.class, c)
```

Unwrapping of adapters is also performed for the implicit upward casts in assignment conversions. In particular, it is always possible to `unwrap` an object when assigning it to an expression of declared type `Object`. For example, the following code

```
I i = new C();  // create adapter
Object o = i;   // adapter not needed, toss it
C c = (C) o;    // no action required
                // because (o instanceof C)
if (i==c)       // true
```

is translated into

```
I i = (I)Registry.convert(I.class, new C());
Object o = unwrap(i);
C c = (C) o;
if (unwrap(i)==c)
```

Class `Registry` can be implemented as a static table that records the adapter classes for each registered occurrence of retroactive abstraction. For implementing `conforms(c,i)`, we search whether there is a chain of types $t_0, \ldots, t_n$, where $\mathtt{c} = t_0$, $\mathtt{i} = t_n$, and t_i is a (retroactive) subtype of t_{i+1}. For testing whether t_i is a retroactive subtype of t_{i+1} it is not necessary to perform the full type check; since only retroactive subtype relationships are allowed that were checked at compile time, it is sufficient to look up the table. Results of this search can be memoized to speed up future searches. Similarly, for implementing `convert(i,c)`, we find the chain of subtypes as above and insert the adapters found in the table for retroactive subtype relationship.

The disadvantage of this translation scheme using adapters and the registry is that it does not work for native methods. The author of native methods must manually write the same code for unwrapping adapters and for using the registry as what is inserted by the compiler. It is not possible, however, to modify existing native methods, such as those used for implementing `java.lang.Thread`. Since most existing native methods, however, do not operate on user objects and have no need to compare them for equality or to perform retroactive subtype tests, we believe that this will not be a problem in practice. Similarly, there are potential problems when passing an object wrapped with an adapter to an existing library method. While an implementation that modifies the virtual machine is semantically cleaner and works for native methods, we believe that using adapters is more desirable in practice than modifying the virtual machine.

5.2 Multimethods

Code generation for generic functions and their multimethods is illustrated by this example:

```
class A {
    generic int foo(I);
    int foo(I x)  { /* 1 */ }
    int foo(J x)  { /* 2 */ }
    int foo(null) { /* 3 */ }
}
```

This is transformed into ordinary Java code, with the exception of the method names, which are not legal Java identifiers, but can nevertheless be represented in Java bytecode with the names shown here.

```
class A {
    int foo(I x) {
        if (x == null)
```

```
            return <multi>_foo_<null>();
        else if (x instanceof J)
            return <multi>_foo((J) x);
        else
            return <multi>_foo(x);
    }
    int <multi>_foo(I x) { /* 1 */ }
    int <multi>_foo(J x) { /* 2 */ }
}
```

For code generation, multimethods are simply renamed by prefixing their name with `<multi>_`, effectively making them inaccessible to ordinary class members. The renamed methods are given `private` access if the generic function has `private` access; `protected` access if the generic function is either `public` or `protected`; or default (package) access if the generic function has default access, so that they can be overridden in a subclass. Recall that we said above that multimethod overriding behaves the same as ordinary method overriding in Java; here we see that it actually is the exact same thing.

The generic function declaration is transformed into a method with the same name and signature that carries out the symmetric dispatch based on the run-time argument types. There are many ways to perform this form of dispatch [14], but for purposes of exposition we show a simple way to do this using `if` statements and `instanceof` tests. In general, though, it is possible to generate more efficient code than a cascading sequence of `instanceof` tests [11] when dispatching on multiple arguments.

The dispatch method corresponding to the generic function must be generated in the generic function's base class and overridden in any subclass that adds new multimethods.

Due to the presence of retroactive abstraction and the adapter classes described above the generated dispatch code might include some redundancies. Consider the following code snippet, where `D` is a subclass of `C` and `I` has been retroactively declared to abstract over class `C`:

```
class B {
    generic int foo(I);
    int foo(I x) { /* 1 */ }
    int foo(C x) { /* 2 */ }
    int foo(D x) { /* 3 */ }
}
```

We would first generate the following source code for the dispatch method:

```
int foo (I x) {
    if (x instanceof D)
        return <multi>_foo((D) x);
    else if (x instanceof C)
        return <multi>_foo((C) x);
    else
        return <multi>_foo((I) x);
}
```

Then the translation for retroactive abstraction would turn this into:

```
int foo (I x) {
    if (x instanceof Ad@pter_C_I && ((Ad@pter_C_I) x).instance instanceof D)
        return <multi>_foo((D) ((Ad@pter_C_I) x).instance);
    else if (x instanceof Ad@pter_C_I && ((Ad@pter_C_I) x).instance instanceof C)
        return <multi>_foo((C) ((Ad@pter_C_I) x).instance);
    else
        return <multi>_foo((I) x);
}
```

However, the above code contains redundant checks and casts. Since retroactive abstraction must be declared explicitly, the new subtype relations it introduces are known at compile time. This information can be used to generate more efficient code like the following version of the dispatch method:

```
int foo(I x) {
    if (x instanceof Ad@pter_C_I) {
        C c = ((Ad@pter_C_I) x).instance;
```

```
            if (c instanceof D)
                return <multi>_foo((D) c);
            else
                return <multi>_foo(c);
        } else
            return <multi>_foo(x);
    }
```

5.3 Binary compatibility

Since multimethods are renamed and made inaccessible, changes to the multimethods, including removal of a multimethod, do not affect binary compatibility, in the following sense: client code that calls the generic function will link without error after the multimethods were changed (provided the package containing the generic function was re-checked). We consider this to be an improvement over static overloading, since removing one method among a set of methods that share an overloaded name does not preserve binary compatibility.

Evolution of packages is restricted by the package-level test: if a new class is added to an existing package, we must check that it does not implement multiple interfaces that would render existing multimethods ambiguous. A change as simple as adding an `implements` declaration to a class, without changing the class body, will trigger a re-checking of a sealed package.

6 Related Work

Multimethod dispatch is found in Common Lisp [30, 33], Dylan [32, 17], and Cecil [10, 12, 9]. These languages, however, only provide symmetric multimethods, in which generic functions dispatch on all arguments symmetrically. Since there is no notion of a receiver, multimethods are outside classes, which causes a lack of encapsulation. Single-dispatch languages such as C++ [16], Smalltalk [20], or Java [21] provide better encapsulation, but do not provide the flexibility of multimethods.

Recently, there have been several attempts at adding multimethods to Java to provide the best of both worlds: parasitic methods [6], TupleJava [26], and MultiJava [13, 14].

Like our multimethods, Boyland and Castagna's parasitic methods are encapsulated multimethods [8, 7]. Parasitic methods are multimethods that can attach themselves to any regular method, which then becomes a generic function. Since an argument type of a parasite can be a supertype of the corresponding argument type of the host method, this solution is semantically not as clean as our solution with a declaration for the generic function. Also, parasitic methods allow parameter types to be interface types without restricting these interfaces to be package-visible, which would require a global link-time multimethod-uniqueness type check.

TupleJava was intended to be an extension of Java with a tuple type [26]. Instead of dispatching on method arguments, a tuple is used as the receiver of a method. This approach has the disadvantage that the client has to know which arguments are used for dispatch. If an operation is added that requires a dispatch on an additional argument, the tuple type and the client have to be modified. Also, tuples require a global link-time multimethod-uniqueness type check.

MultiJava uses Millstein and Chambers' type restrictions [29] for allowing the language to be statically typed. To allow a static multimethod-uniqueness type check, a multimethod cannot be dispatched on interface types, which results in a loss of expressiveness. MultiJava uses symmetric multimethod dispatch that treats the receiver the same as other arguments. For implementing the Visitor pattern using symmetric multimethods, they introduce open objects, that allow external methods to be added to existing classes without defining a subclass. This causes the semantics of method dispatch on the receiver to change without a corresponding syntactic change, which makes client code harder to debug. For a given method call, it is no longer obvious where to find the method that will be executed; it may not be in the class of the receiver anymore. Another problem with MultiJava is that by allowing methods to be dispatched on some arguments but overloaded on others, the method selection algorithm becomes unnecessarily complicated. Like with overloading in Java, method selection consists of compile-time overload resolution and a run-time dispatch.

Retroactive abstraction is supported by Sather [34]. In this language, the definition of a new abstract class may include a supertyping clause, which defines the new abstract class as the supertype of one or more existing classes. However, once an abstract class has been defined, it cannot be used as the supertype in later retroactive abstraction.

Aspect-oriented programming [23, 22] has been proposed as a technique for improving the separation of concerns in software. For example, aspects may add new methods (and fields) to existing classes without modifying the original definitions of those classes. To support this capability, aspect-oriented programming requires access to the source of the entire program to insert the aspects into the affected classes. Therefore, changing an aspect requires recompilation of all classes extended by this aspect. While Half & Half does not handle general cross-cutting concerns as well as aspect-oriented programming, it does accommodate components available only in compiled form and, thus, meets the requirements of a solution to the extensibility problem as defined in Section 2.

7 Conclusions

We have presented Half & Half, a conservative extension of Java with multimethod dispatch and retroactive abstraction. The combination of multimethods and retroactive abstraction allows many software design problems, including the expression problem, to be solved elegantly and simply.

Multimethods in Java have an additional advantage over overloading. According to the Java rules for binary compatibility [21], clients of a class do not have to be recompiled when adding a method to the class. With overloading, a certain method signature was selected when compiling a method call in the client. If a more specific method is added later, the client would continue to call the old method. When recompiling the client, the more specific method will be chosen. With multimethods, the compiled client does not have the method descriptor (signature and return type) of the old method wired in, but the descriptor of the dispatch method. Even if there is a more specific method, the old dispatch method still applies.

We have integrated multimethod dispatch with retroactive abstraction by allowing multimethods to be dispatched on interface types. We have described a type system restriction that allows multimethods to be type-checked with only a package-wide type check equivalent to link-time type checking.

We have presented an implementation strategy that is compatible with the existing Java Byte Code format and with existing Java virtual machines. We are currently implementing Half & Half as a front end of the Brew compiler.

References

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[2] Gerald Baumgartner, Martin Jansche, and Christopher D. Peisert. Support for functional programming in brew. In *Proceedings of the 2001 Workshop on Multiparadigm Programming in Object-Oriented Languages*, Budapest, Hungary, June 2001.

[3] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Sciences, Purdue University, February 1996.

[4] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software—Practice & Experience*, 25(8):863–889, August 1995.

[5] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[6] John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. In *Proceedings of the OOPSLA '97 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 66–76, Atlanta, Georgia, 5–9 October 1997. Association for Computing Machinery. *ACM SIGPLAN Notices*, 32(10), October 1997.

[7] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[8] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[9] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehr mann Madsen, editor, *Proceedings of the ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Utrecht, The Netherlands, 29 June – 3 July 1992. Springer-Verlag, Berlin, New York.

[10] Craig Chambers. The Cecil language: Specification and rationale: Version 2.0. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, December 1995. `http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-s%pec.html`.

[11] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the OOP-SLA '99 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 238–255. Association for Computing Machinery, October 1999. *ACM SIGPLAN Notices*, 34(10), October 1999.

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

[13] 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 archives.cs.iastate.edu.

[14] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the OOPSLA '00 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, 15–19 October 2000. Association for Computing Machinery.

[15] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, NY, 1991.

[16] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[17] Neil Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Massachusetts, 1997.

[18] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1998.

[19] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[20] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 2nd edition, 2000.

[22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[24] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113, 1998.

[25] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *Computer Journal*, 43:469–481, 2000.

[26] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of the OOPSLA '98 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 374–387, Vancouver, British Columbia, Canada, 18–22 October 1998. Association for Computing Machinery. *ACM SIGPLAN Notices*, 33(10), October 1998.

[27] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings of the OOPSLA '98 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 36–44, Vancouver, British Columbia, Canada, 18–22 October 1998. Association for Computing Machinery. *ACM SIGPLAN Notices*, 33(10), October 1998.

[28] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[29] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the 1999 European Conference for Object-Oriented Programming (ECOOP '99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, 14–18 June 1999. Springer-Verlag.

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

[31] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay, France, 1975.

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

[33] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, Massachusetts, 2nd edition, 1990.

[34] David Stoutamire and Stephen Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, ICSI, Berkeley, CA, 1996.

[35] Sun Microsystems, Inc. API specification for the Java 2 platform, standard edition, version 1.3. Available at `http://java.sun.com/j2se/1.3/docs/api/index.html`, 2000.

[36] Philip Wadler. The expression problem. Posted to the Java-Genericity Mailing List, 12 November 1999.

[37] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. International Conference on Functional Programming*. ACM Press, Florence, Italy, September 2001.

[38] Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *Proc. ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.