OOMatch: Pattern Matching as Dispatch in Java

Adam Richard
University of Waterloo
a5richard@uwaterloo.ca

Ondřej Lhoták
University of Waterloo
olhotak@uwaterloo.ca

Abstract

We present a new language feature, specified as an extension to Java. The feature is a form of dispatch, which includes and subsumes multimethods (see for example [CLCM00]), but which is not as powerful as general predicate dispatch [EKC98]. It is, however, intended to be more practical and easier to use than the latter. The extension, dubbed OOMatch, allows method parameters to be specified as *patterns*, which are *matched* against the arguments to the method call. When matches occur, the method applies; if multiple methods apply, the method with the *more specific* pattern *overrides* the others.

The pattern matching is very similar to that found in the "case" constructs of many functional languages (ML [MTHM97], for example), with an important difference: functional languages normally allow pattern matching over *variant* types (and other primitives such as tuples), while OOMatch allows pattern matching on Java objects. Indeed, the wider goal here is the study of the combination of functional and object-oriented programming paradigms.

Maintaining encapsulation while allowing pattern matching is of special importance. Class designers should have the control needed to prevent implementation details (such as private variables) from being exposed to clients of the class.

We here present both an informal "tutorial" description of OOMatch, as well as a formal specification of the language. A proof that the conditions specified guarantee run-time safety appears in the companion thesis [Ric07].

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects, Patterns, Polymorphism, Procedures, functions, and subroutines

General Terms Design, Languages

Keywords predicate dispatch, dynamic dispatch, pattern matching, multimethods, Java

1. Introduction

This paper presents a small step towards the goal of unifying object-oriented and functional programming. In particular, it considers how pattern matching – a common and useful feature of many functional languages – might be interwoven into the object-oriented tapestry. Pattern matching in, for example, ML, allows one to decompose algebraic types or tuples into their components, either in a case statement or in a set of functions. Though this pattern matching is useful in a functional context, simple algebraic types and tuples are not used much in object-oriented programming; classes are used much more. So we here present a means of deconstructing objects into their components and specifying patterns that match objects with certain properties.

The most significant difference from pattern matching in functional languages is that in OOMatch, pattern matching is used in determining method dispatch. Thus, pattern precedence cannot be determined by textual order, as it is in functional languages. The patterns are specified as parameters to methods, and the compiler decides on a natural order to check for a matching pattern, i.e. to check which methods override which. Methods with more specific parameters override methods with more general parameters. A key contribution of our work is a careful definition of which patterns are "more specific" than others. Since parameters of subclass type are considered more specific than those of superclass type, this feature subsumes polymorphic dispatch and multimethods. Information hiding of objects (a fundamental property of object-oriented systems) must be preserved; we must not allow clients to access the data of the object except in ways the class writer explicitly allows. Another important goal is simplicity; if programmers find the facility confusing, they can simply use if-else blocks and casting instead of pattern matching. The practical value of pattern matching as dispatch would then be lost.

The matching is done with the aid of special methods, called *deconstructors*, which return the components of an object in a way that the class designer has control over. To enable pattern matching on an object, the programmer writes a deconstructor for the object. Alternatively, there is a syntactic sugar that allows a constructor and deconstructor to be written at once, and is sufficient for many cases.

OOMatch has been implemented in the Polyglot Extensible Compiler Framework [POL]. Polyglot translates to Java, but contains all the functionality of compiling the base Java language, which prevents implementers from having to write a compiler from scratch. It is therefore useful for writing prototype compilers for new Java-like languages. The implementation is described in detail in [Ric07], and can be downloaded from http://plg.uwaterloo.ca/~a5richar/oomatch.html.

The paper is organized as follows. Section 2 gives background and related work leading up to OOMatch and should be helpful to those unfamiliar with all the concepts used. Section 3 gives an informal description of the language and its various features. Section 4 gives a formal specification of the core of OOMatch (the pattern matching dispatch). It also describes the checks done by the compiler, and claims that if an OOMatch program compiles, there are only certain given conditions under which a run-time error can occur. Section 5 briefly reports some of our experience in refactoring Java code to OOMatch. Section 6 discusses other related work that is best discussed after the reader understands OOMatch. Section 7 concludes.

2. Background

OOMatch combines two broad areas of programming languages research - pattern matching and dispatch mechanisms. We describe each of these areas in turn, along with research in those areas related to or leading up to OOMatch.

2.1 Pattern Matching

Pattern matching is a popular feature in functional languages. Suppose a programmer wants to test whether the first element of a pair is 0, and if it is, return the second element. In a language without pattern matching, such as Java, this can be achieved with the following code:

```
if (pair.first() == 0) {
    return pair.second();
}
else { ... }
```

In other words, accessor functions and comparisons are necessary to get the components of a structure and test them for the relevant conditions.

In a language with pattern matching, such as SML [MTHM97], the following simpler code accomplishes the same task:

```
case pair of
    (0, second) => second
|   _ => ...
;;
```

Pattern matching involves taking an expression (`pair` in this case) and a pattern (`(0, second)` in this case), and trying to find a set of substitutions for variables in the pattern such that it is equal to the expression. If such a set of substitutions exists, the match succeeds; otherwise, it fails, and in the case of the `match` construct of ML (above), the next case is tested. Often, one of the patterns contains one or more *free variables*, which normally act as a wildcard for matching purposes. In the example above, `second` is a free variable. When a match succeeds, the free variables normally receive a unique value that was necessary to do the match, and those variables can be used in some succeeding block of code (the case of the `match` construct, in this example). To use an analogy from mathematics, pattern matching is like giving the compiler an equation and letting it solve for the variables, rather than programmers solving the equation themselves.

Whereas the pattern matching in ML, shown above, operates on built-in language constructs (tuples in this case), a more challenging problem is how to allow matching of objects. Object matching is tricky because it involves decomposing objects of a class into components, and it is not obvious to the compiler what the components of an object are, as far as the writer of the class is concerned. Further, the class writer may not want clients to have access to those components, or may want to only allow access to them in a controlled way. OOMatch provides this control with the use of special functions called *deconstructors*, described later. Other languages with object-oriented pattern matching, described in the following subsections, have similar approaches to this problem.

2.2 Dispatch

Method dispatch means the way in which a language determines, given a call site, which method to call. In the days of the original Fortran, this task was simple, because each function in these early languages generally had a unique name - the name in the call site hence uniquely determined a method, and this method could be fixed at compile time.

Later, languages began to allow multiple functions with the same name. When multiple such methods are present, there are two main ways in which the correct method to call can be determined - by using only the static type information of the arguments passed in the method call (overloading), or by using the run-time type information, and determining the method at run-time (overriding).

Overloading is the presence of multiple methods with the same name and in the same class hierarchy. The different parameters (and possibly different return values) among the methods allow the compiler to determine which method to call given a call site - it finds the static types of the method arguments and chooses the method whose parameters correspond to those types. There may be various rules regarding what to do if multiple methods are eligible, depending on the language. Again, the method to call is fixed at compile-time. Overloading is convenient because it allows programmers to provide several ways of invoking what is conceptually the same operation.

Many object-oriented languages, including Java, have a feature called *receiver-based* dispatch, which means the method selected at a call site can change at run-time, depending on the actual (dynamic) type of the receiver argument it is called on. It is also called *dynamic dispatch* because the callee is chosen dynamically (not until run-time).

Receiver-based dispatch is very useful for data abstraction. For example, suppose a programmer had a variable representing a shape, and wanted to call a method to draw it:

```
Shape s;
...
s.draw();
```

Because `s` could be one of many kinds of shapes - polygon, circle, etc. - it would be inconvenient to have to write a `draw` function that can draw any of them, depending on what kind of shape `s` is. Further, there might be other kinds of shapes the programmer who wrote the above call did not know about. With receiver-based dispatch, the programmer can instead write one `draw` method for each type of `Shape`:

```
class Shape {
    public void draw() {}
}

class Circle extends Shape {
    public void draw() {  //overrides Shape.draw
        //draw a circle
    }
}

class Rectangle extends Shape {
    public void draw() {  //overrides Shape.draw
        //draw a rectangle
    }
}
```

Now, if `s` is a `Circle`, `s.draw()` invokes `Circle.draw` - even though the static type of `s` is `Shape` - because `Circle.draw` overrides `Shape.draw`.

There has been research on other, more powerful forms of dynamic dispatch, which subsumes receiver-based dispatch. A sample of this research is discussed next.

2.2.1 Multimethods

Multimethods are a classic example of a powerful form of dispatch. They were introduced in CommonLoops [BKK$^+$86] and added to Java in MultiJava [CLCM00]. They allow the method chosen to depend on the run-time types of *all* arguments, rather than just the receiver argument. For example, consider these two methods:

```
class C {
    Shape intersect(Shape s1, Shape s2) { ... }
    Shape intersect(Circle s1, Square s2) { ... }
}
```

In Java, of course, these methods would be overloaded. With multimethods, the second method would instead override the first method, so that if `intersect` is called with a pair of `Shape` variables that are really a `Circle` and `Square`, respectively, at run-time, the second method takes precedence and is called.

To understand the usefulness of this feature, consider how one might write a class with an "equals" method. In Java, a naive programmer might write the following:

```
class C {
    ...
    public boolean equals(C other)
    { ... }
}
```

But this is incorrect because the version of "equals" shown does not in fact override Object's "equals" method, which has signature `public boolean equals(Object obj)` [JAV]. Because the `equals` in `C` does not have the same parameter types, the methods become overloaded rather than overridden. This means that, for example, this code:

```
Object o = new C(...);
if (new C(...).equals(o)) {...}
```

does not call the user's `equals` method, but the one in the Object class, probably causing unexpected behaviour. In an imaginary language where the methods were treated as multimethods, `C.equals(C)` would override `C.equals(Object)` which would in turn override `Object.equals(Object)`, and the behaviour that was probably expected would take place. Instead, in Java, one must (and must remember to) write custom dispatch code, such as:

```
class C {
    ...
    public boolean equals(Object otherObject)
    {
        if (!(otherObject instanceof C))
            return false;
        C other = (C)otherObject;
        ...
    }
}
```

This code is noticeably more verbose and error-prone than the multimethod version.

The Visitor design pattern [GHJV94] is a way to simulate double dispatch (i.e., multimethods on only the class parameter and one explicit parameter) in an Object-oriented language with only regular polymorphic dispatch. Multimethods obviate the need for visitors, and are also more general than visitors, since they can dispatch on more than two parameters.

2.2.2 Predicate Dispatch

The notion of multimethods was further generalized, and formalized as predicate dispatch, in [EKC98]. In predicate dispatch, any arbitrary predicate can be used to choose the method to call. The idea is that a boolean condition is added to a method definition, and when the condition evaluates to true (at the time of a method call), that method is called. If a method A's condition implies B's (where A and B have the same name and argument types but different boolean conditions), then A is said to override B.

While predicate dispatch is an excellent aid in understanding and motivating various forms of dispatch, we would like to provide the common programmer with a language feature that is less powerful but easier to use. In particular, it is cumbersome to extract the internals of objects when using general predicate dispatch, involving dereferencing and comparisons in the boolean predicate. Perhaps more importantly, doing so requires the data members of objects to be exposed, which violates encapsulation.

Another tradeoff that full predicate dispatch necessarily makes is that few safety guarantees can be made at compile-time. If the boolean predicates can contain arbitrary code, it is impossible for the compiler to tell, in general, whether one condition implies another. Hence, it cannot ensure that there will not be multiple methods applicable to a call, which leads to crashes or unexpected behaviour at run-time. Hence, while predicate dispatch is by definition the most powerful form of dispatch, we believe there is a "sweet spot" somewhere between it and multimethods, which is less error-prone and can resolve many of these ambiguities in a known, practical way.

3. Using OOMatch - Informal Description

3.1 Pattern Matching

We introduce OOMatch using a simple example. Suppose one is writing the optimizer component of a compiler, and wants to write code to simplify arithmetic expressions. Suppose the Abstract Syntax Tree (AST) is represented as a class hierarchy (a natural way to represent an AST), as follows.

```
//Arithmetic expressions
abstract class Expr { ... }

//Binary operators
class Binop extends Expr { ... }

//'+' operator
class Plus extends Binop { ... }

//Numeric constants
class NumConst extends Expr { ... }

//Integer constants
class IntConst extends NumConst { ... }
```

Then part of the functionality to simplify expressions could be implemented using OOMatch as the following set of methods:

```
//do nothing by default
Expr optimize(Expr e) { return e; }

//Anything + 0 is itself
Expr optimize(Plus(Expr e, NumConst(0)))
{ return e; }

//Constant folding
Expr optimize(Binop(NumConst c1,
                    NumConst c2) op)
{ return op.eval(c1, c2); }
```

These methods are matching appropriate types of expressions and applying optimizations when possible. Each method specifies an optimization rule. The latter two methods, which also have one parameter each, specify patterns to break down or "deconstruct" that parameter into its components, which are matched against the argument passed to `optimize`. The second method, for example, takes a parameter of type `Plus` and breaks it into two parts (the two operands of the "+" operator), `Expr e` and `NumConst(0)`. That method only applies, then, when the argument is of type `Plus`, and the operands match these two patterns. We assume that all operands are of type `Expr`, so the first operand always matches, while the second one apparently matches when the other operand is a numeric constant with the value 0. Note that, to be able to write the patterns shown, the classes being matched against (`Plus`, `Binop`, and `NumConst` in this case) require support to allow them to be matched. The way this support is provided is described shortly, in Section 3.2.

The key point to notice in the above example is that the second method *overrides* the first, since its pattern is *more specific* (because `Plus` extends `Expr`), and the third also overrides the first since `Binop` extends `Expr`. Note that the order in which the methods appear does not affect these override relationships.

The `0` in the second method means that the pattern is only matched when the numeric constant's value is 0. The named variables in the patterns are given the value that is matched, so that this value can be used by referring to the declared name in the method body.

The entire object represented by a pattern can be given a name. In the example above, the Binop in the third method is given the name "op" so that the matched object can be referred to in the method.

Patterns can of course themselves contain patterns (as is shown in the second method above), and can indeed be nested to an arbitrary depth. The most specific match is always chosen first. So, for example, we could add another method with signature

```
Expr optimize(Binop(IntConst c1, IntConst c2))
```

This new method overrides the third one, because the pattern type is the same but the subpatterns are more specific.

Note that OOMatch introduces the potential for new kinds of errors. In fact, the above code contains such an instance. If `optimize` is passed an expression like `1 + 0`, the second and third methods both apply, because this expression is both adding 0 to an expression and performing an operation on two constants. However, it cannot be said that either of these methods overrides the other, because there are cases where the second applies and the third does not, and vice versa. This is called an ambiguity error — it is possible for more than one method to apply, but neither is necessarily more specific than the other. Normally, this results in a compile error, though there are cases where the compiler cannot detect ambiguity errors, as we shall see later. In this case, the problem could be resolved by adding a fourth method which handles the intersecting case:

```
Expr optimize(Plus(NumConst e, NumConst(0)))
{ return e; }
```

The other new kind of error that can be present in an OOMatch program is when no method can be found for a call site: this is called a no-such-method error. Normally, the compiler prevents these by requiring that all methods with patterns override a method with only regular Java formals, either in the same class or a superclass. In this way, the regular Java method can always be called as a last resort.

For example, consider the following method:

```
void f(NumConst(0)) { ... }
```

If this method appeared alone, it would result in an incomplete error, because the case `NumConst(1)` (among others) is not handled.

However, sometimes the programmer either does not care about this assurance or wants to use patterns as a form of preconditions (as in the D programming language [D], for example), requiring that the arguments to a method have a certain form and giving a runtime error if they do not. For these cases, OOMatch allows methods to be labelled with the keyword `inc`, for incomplete. A method labelled `inc` will not cause a compile error if it does not override anything, but might cause a no-such-method error at runtime.

The two errors are of type `java.lang.Error` when thrown. Catching and handling them is possible, but is usually considered bad style.

3.2 Deconstructors

To allow the specification of patterns on objects, as in the previous section, their classes must provide a means of *deconstructing* said objects. There are two ways of doing so in OOMatch. The first way, described next, is simplest but allows little control; the second option allows the class writer much greater control over access to the class.

In OOMatch, instance variables can be declared within constructor parameters:

```
class Binop {
    public Binop(public Expr e1,
                 public Expr e2)
    { ... }
    ...
}
```

Adding the `public` access specifier to a parameter has three effects:

- The variable becomes a public instance variable of the class. If there already is an instance variable of that name, a duplicate definition error occurs. The specifiers `private` and `protected` have similar effects; the generated instance variable is given the access permissions given by the specifier.
- The argument passed to the constructor is assigned to the instance variable. This assignment happens at the *end* of the constructor body.
- A deconstructor is created that allows the object to be pattern-matched in a pattern that corresponds to the way it was constructed. Constructor parameters with any access specifier become deconstructor parameters.

If there are multiple constructors with the same instance variables declared as parameters, no error occurs; they each construct the same variable. Likewise, it is no problem to have two constructors with different instance variables; all parameters become instance variables, but automatic assignments only take place for the constructor that is called.

Deconstructing an object means that certain components of the object are being "returned", and then matched against. So for

```
Expr optimize(Binop(NumConst c1,
                    NumConst c2) op)
```

the instance variables e1 and e2 are extracted from the `Binop` argument, and if they are both instances of NumConst, they are assigned, by reference, to the variables c1 and c2.

The above syntax is convenient and intuitive because objects can be deconstructed in the same way they were constructed. Moreover, even in the absence of pattern matching, the ability to write both instance variables and constructor parameters all at once provides a handy shortcut for writing quick-and-dirty classes for which access is not important. But in large object-oriented systems, it is crucial that programmers are able to restrict access to data members. Hence, the more general and powerful notion of a *deconstructor*, described next, is provided.

An equivalent way to write the Binop class in OOMatch is as follows.

```
class Binop {
    public Expr e1, e2;
    public Binop(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
        ...
    }
    deconstructor Binop(Expr e1, Expr e2)
    {
        e1 = this.e1;
        e2 = this.e2;
        return true;
    }
    ...
}
```

A deconstructor breaks down `this` into components, and returns them to be matched against. But rather than returning said components in the return value, its parameters are "out" parameters, each one representing a component. The deconstructor must assign each of them a value on each possible path through its body; they have no defined values at the beginning of the body. This rule is enforced conservatively with the same analysis that Java uses to enforce initialization of variables. Aside from these restrictions, any arbitrary code may appear in a deconstructor, and any values of type `Expr` can be returned in the parameters `e1` and `e2` in the example above. This way class writers can restrict access to instance variables (by making them private, etc.), while still being able to use them in pattern matching.

A deconstructor must always return a boolean value, which indicates whether the match was successful. This allows even patterns that would otherwise match to fail (by returning false) under certain arbitrary conditions, such as the state of the object. For example, perhaps one wants to prevent matching a file object when the file has not been opened yet.

Of course, in a real-world application, the instance variables above would probably be private and accessed using accessor methods. Indeed, this is exactly what deconstructors allow one to do.

Note that the (perhaps confusing) syntactic notation of deconstructors returning their values in "out" parameters is necessary because Java lacks multiple return values. A more elegant, and understandable to the user, syntax would be for deconstructors to return a tuple of values, which supposedly represent the components of `this`. Any method which takes no parameters and returns a tuple could then be used as a deconstructor. This approach was taken by Scala's extractors [EOW07], for example.

In general, method headers in OOMatch can contain regular formal parameters, or patterns. Patterns can contain literal primitive values (including string literals), but there is no way to put literal objects in a pattern. In other words, one cannot specify a "new" expression in a parameter to match against. They can, however, provide a deconstructor for the object and specify a specific object as a pattern with specific subcomponents. Also note that literals can appear by themselves, in place of regular parameters. For example, the following pair of methods is allowed (and is potentially useful):

```
void f(int x) { ... }
void f(0) { ... }
```

The second method above overrides the first.

Note also that a deconstructor can be given any name, not just the name of the class. If given a name other than the name of the class, any references to the deconstructor must be prefixed with the class name, as in:

```
Expr optimize(Expr.my_deconstructor(
    NumConst c1, NumConst c2))
{ ... }
```

From the point of view of the OOMatch compiler, referring to a deconstructor as `X.Y` is the desugared form, and means that a deconstructor named `Y` is looked up in the class `X` or its superclasses. When a deconstructor is referred to as simply `X`, the compiler first looks for a deconstructor `X` in the class `X`; if none is found, it looks in the superclass of `X` for a deconstructor with the name of the *superclass*, and so on for each superclass. Hence, the expression `Plus(Expr e, NumConst(0))` seen earlier could be short for `Plus.Binop(Expr e, NumConst(0))` - meaning a value of type `Plus` deconstructed with a deconstructor in its superclass, `Binop` - if the class `Plus` does not have a deconstructor of its own. This syntactic sugar is meant to coincide with the syntactic sugar for combined constructor-deconstructor, so that simply specifying the deconstructor as `Plus` is saying, "deconstruct an object of type `Plus`".

3.3 Order of Deconstructors

When determining which method applies to a method call, deconstructors must sometimes be called. The order in which they are called is left unspecified. This choice was made to free implementations to do optimizations that may require certain implementations of the dispatch algorithm. Further, implementations may choose not to run the deconstructor for a given pattern, as long as the required dispatch semantics are preserved. However, we do make the requirement that, for a given method call, a deconstructor is run at most once for each reference to it.

Because deconstructors are not intended to have side effects, it is not normally useful to write code which depends on the deconstructors that are called and the order in which they are called. Hence, this implementation-defined behaviour was deemed more desirable than explicitly-defined behaviour, because it increases the potential for optimizations.

3.4 Null

Null parameters introduce some interesting cases. First, null literals override any formal parameter of class type. Suppose there are two classes `A` and `B`, unrelated by inheritance, and this class:

```
class C {
    void f(A a) { ... }
    void f(B b) { ... }
    void f(null) { ... }
}
```

The third method overrides both the others. There is no way to specify that one is matching only null values of a particular static type; syntax to allow this could be a possible future addition. Otherwise, `null` is doing nothing special here; since `null` is a value of all class types, it overrides all methods with a single parameter of class type, as expected.

Another trickier issue with `null` is that it cannot be deconstructed. Given the `Binop` deconstructor from Section 3.2, one might expect the following lone method to present no problems, as it handles every `Binop` object:

```
class C {
    inc void f(Binop(Expr e1, Expr e2))
    { ... }
}
```

But unlike a method that takes a single parameter of type `Binop`, this one cannot be passed the value `null`, because `null` cannot be deconstructed. If this is attempted, a run-time error occurs.

3.5 Undecidable Errors

Though the compiler can detect many of the new ambiguity and no-such-method errors statically, finding all of them is undecidable. Rather than restricting the language and disallowing certain programs that make sense, we have chosen to throw an exception at run-time when the ambiguities described here occur.

3.5.1 Interfaces

The first potential cause of ambiguity is caused by multiple inheritance, which is partially allowed in Java for interfaces.

```
void f(A a) { ... }
void f(B b) { ... }
```

where `A` and `B` are interfaces that are not related at all. Despite this being entirely valid Java, the compiler cannot guarantee that this program is free from ambiguity errors, because it might happen that there is a class `C` which implements both `A` and `B`, and if an object of type `C` is passed to `f`, OOMatch does not know which version to call. It is not possible to tell whether such a `C` exists at compile time; not only does separate compilation preclude knowledge of all the subclasses of `A` and `B`, but dynamic class loading means that knowing what classes will be present at the time of a call to `f` is, in general, undecidable.

To fix this problem when it arises, a programmer can simply use a cast to disambiguate the method call:

```
C o;
...
f((A)o);
```

This causes `f(A)` to be chosen and `f(B)` to be removed from consideration. It works because dispatch requires the argument to be either a subclass or a superclass of the parameter type for the method to be chosen, and `A` is neither a subtype nor supertype of `B`.

3.5.2 Different deconstructors

The next type of ambiguity can occur when there is a pair of methods that could be called from a call site, and a corresponding parameter is referring to a different deconstructor in each method. For example, let us take the `Binop` class from before and add an extra deconstructor to it:

```
class Binop {
    ...
    deconstructor Binop(Expr e1, Expr e2)
    { ... }
    deconstructor Binop2(Expr e1, Expr e2)
    { ... }
}
```

Now suppose we have a set of methods that matches on both of them:

```
class C {
    void f(Binop(Expr e1, Expr e2)) { ... }
    void f(Binop.Binop2(Expr e1, Expr e2)) { ... }
}
```

Since both patterns appear to match every `Binop`, it may at first appear that this is clearly an ambiguity, or even a duplicate method definition. But in fact it is not necessarily so. Since deconstructors can run arbitrary code and return `true` or `false` depending on whether they match, it is quite possible for the programmer to ensure that they match only in a mutually exclusive manner. For example, the Binop class could keep track of a boolean flag and only match one deconstructor when it is true, and the other when it is false. But the compiler cannot decidably determine whether they will both match in some cases. So, to ensure that it allows all programs that make sense, we have decided to wait until run-time to give the error in this case.

Note that it makes no difference whether the pattern contains constants in its parameters or not, or whether one pattern appears to be more specific than the other. Since the deconstructors may be returning completely different values, (there is no rule forcing them to return instance variables of the class, for example) the compiler can say nothing about whether both methods always apply simultaneously, whether they are mutually exclusive, or whether one overrides the other. Hence, it assumes they are mutually exclusive, and a run-time error occurs if this turns out not to be so.

3.5.3 Non-deterministic deconstructors

Finally, because deconstructors can return any values, problems can arise if they return different values on different invocations. Consider the following pair of methods which use the class `Point` described above:

```
class C {
    void f(Point(0, 0)) { ... }
    void f(Point(1, 1)) { ... }
}
```

It may appear that these methods are clearly mutually exclusive. But in fact, nothing prevents the deconstructor for `Point` from being implemented like so:

```
    deconstructor Point(int x, int y) {
        Random r = new Random();
        //Randomly return either 0 or 1
        //for each of x and y
        x = r.nextInt(2);
        y = r.nextInt(2);
    }
```

In this case, it is quite possible that on the first invocation of the deconstructor, two zeroes are returned, and on the second invocation, two ones are returned, which makes both methods match. In general, a deconstructor should have no side effects, and always return the same set of values given the same objects. Again, the compiler cannot determine, in general, whether this is so. In a language with special methods that are not allowed to modify any variables other than those declared in its body, this would become easier. This property could be assured with the help of immutability checking, such as that found in Javari [TE05]. However, due to its complexity, it has been left as future work.

This kind of non-deterministic behaviour is, of course, not very useful in a pattern matching context, and is, hence, relatively easy to avoid; on the other hand, such problems, if they are somehow introduced, could potentially be very difficult to find and debug. On the plus side, this problem, as well as the other two mentioned in this section, could be found with a static program analysis in many cases.

4. Formal Specification

4.1 Syntax

We present the core (desugared) syntax of OOMatch by making two modifications to the Java grammar from Chapters 3 and 8 of the Java Language Specification, second edition [GJSB96]. In this section, we show differences from the Java grammar in bold.

OOMatch adds deconstructors as a new kind of class member:

ClassMemberDeclaration ::= *FieldDeclaration*
| *MethodDeclaration*

| *ClassDeclaration*
| *InterfaceDeclaration*
| ***deconstructor***
deconstructor ::= `deconstructor` *Identifier*
(*FormalParameterList*$_{opt}$) *MethodBody*

In addition to formal parameters as in Java, OOMatch allows methods to have two new kinds of parameters: literals and patterns.

MethodHeader ::= *MethodModifiers*$_{opt}$ *ResultType*
MethodDeclarator Throws$_{opt}$
MethodDeclarator ::= *Identifier* (***OOMatchParameterList***$_{opt}$)
OOMatchParameterList ::= ***OOMatchParameter***
| ***OOMatchParameterList* , *OOMatchParameter***
OOMatchParameter ::= *FormalParameter*
| *Literal*
| ***Pattern***
Pattern ::= *Type* . *Identifier* (***OOMatchParameterList***$_{opt}$)

Because in Java, the floating point literals `-0.0` and `0.0` are considered equal, as are the integer literals `-0` and `0`, OOMatch considers them the same literal. For example, the method signature `void m(0.0)` is considered to be the same as `void m(-0.0)`.

4.2 Notation and Definitions

Throughout this section, we will use the following abbreviations for OOMatch entities.

- $F[T]$ represents a Java formal parameter of type T.
- $C[v, T]$ represents an OOMatch literal parameter with Java literal value v and type T.
- $P[T_r, n, \vec{T_p}]$ represents an OOMatch pattern with type T_r, name n, and parameter types $\vec{T_p}$.
- $D[n, \vec{T_p}]$ represents a deconstructor with name n and out-parameter types $\vec{T_p}$.
- $M[T_r, n, \vec{T_p}, \vec{T_t}]$ represents a method with return type T_r, name n, parameter types $\vec{T_p}$, and declared throw types $\vec{T_t}$.

We explicitly define a subtyping relation that corresponds to the assignability rules defined in the Java Language Specification [GJSB96].

DEFINITION 4.1. *The subtyping relation* $<:$ *is the smallest transitive and reflexive relation satisfying the following:*

1. $T <: T'$ *if* T *and* T' *are classes or interfaces and* T *extends or implements* T'.
2. *null* $<: T$ *if* T *is a class, interface, or array type.*
3. `byte` $<:$ `short` $<:$ `int` $<:$ `long` $<:$ `float` $<:$ `double`, *and* `char` $<:$ `int`.
4. *For array types,* A`[]` $<: B$`[]` *if* $A <: B$.
5. $T <:$ `Object` *for all class, interface, and array types* T.
6. $T[] <:$ `Cloneable` *and* $T[] <:$ `java.io.Serializable` *for all array types* $T[]$.

The following lemma is important to ensure that our notion of parameter preference (below) does not contain cycles.

LEMMA 4.1. [1] *Subtyping is a partial order.*

[1] Proofs of all lemmas and claims can be found in the companion thesis [Ric07].

4.3 Deconstructor Binding

At compile time, every pattern appearing in the program is statically bound to a fixed deconstructor, which will be used to evaluate the pattern. To specify which deconstructor is to be used for a given pattern, we first define the *type* of a parameter as follows:

$$\mathit{type}(F[T]) = T$$

$$\mathit{type}(C[v, T]) = T$$

$$\mathit{type}(P[T, n, \vec{p}]) = T$$

Then, a deconstructor $D[n_1, \vec{T_1}]$ is *eligible* for a pattern $P[T_2, n_2, \vec{p_2}]$ if

- The deconstructor is in T_2 or one of its supertypes
- they have the same name ($n_1 = n_2$),
- they have the same number of parameters ($|\vec{T_1}| = |\vec{p_2}|$), and
- the type of every parameter of the pattern is a subtype of the corresponding parameter of the deconstructor ($\forall i.\mathit{type}(p_{2i}) <: T_{1i}$).

A deconstructor $D[n, \vec{T}]$ is *more specific* than $D'[n', \vec{T'}]$ if every parameter of D is a subtype of the corresponding parameter of D' ($\forall i.T_i <: T'_i$).

The deconstructor bound to a given pattern must be eligible for the pattern, and it must be more specific than every deconstructor eligible for the pattern. A compile-time error is generated when these conditions are not satisfied by any deconstructor.

4.4 Method Invocation

We now specify how OOMatch determines, at a given call site and with specific runtime arguments, which method to invoke. We break the specification into three parts. First, we define a set of methods that are *applicable*, in that they could be invoked provided no "more specific" method is available. Second, we define a partial order on the set of applicable methods to decide which methods shall be preferred over others. Finally, we use these definitions to specify how OOMatch selects the method to be executed.

4.4.1 Applicable methods

The predicate *applicable*$(M[T_r, n_M, \vec{p}, \vec{T_t}], n, \vec{a})$ is defined on a method with return type T_r, name n_M, parameters $\vec{p}$, and throwing types $\vec{T_t}$; the name n of the method to be invoked at a call; and a list of argument values $\vec{a}$. The predicate is true when all of the following conditions hold:

1. The name of the method matches the name at the call site: $n_M = n$.
2. The number of arguments and number of parameters are equal: $|\vec{a}| = |\vec{p}|$.
3. Each argument is *admissible* for its corresponding parameter: $\forall i.\mathit{admissible}(a_i, p_i)$. Admissibility is a generalization of the Java guarantee that a method with a given parameter type is only called with arguments that are instances of that type. The admissibility condition is made precise below.

The predicate *admissible*(a, p) is defined on an argument a of statically declared type T_s and run-time type T_d, and a parameter p. Recall that an OOMatch parameter can be a Java formal, a Java literal, or a pattern.

1. When p is a Java formal and when a is not null, admissibility is determined as in Java: a is admissible if in Java it is method invocation convertible [GJSB96, Section 5.3] to the type declared for p. When a is null, it is admissible if its statically declared type T_s is a subtype of the type declared for p.

2. When p is a literal l, a is admissible exactly when it is equal to l. For string literals, equality is defined as l`.equals(`a`)` returning true; for all other literals, equality is defined as the Java `==` operator.

3. When p is a pattern $P[T_r, n, \vec{p}]$ with type T_r, name n, and parameters $\vec{p}$, OOMatch first checks whether the runtime type T_d of a is a subtype of T_r. If it is not, then a is not admissible. If it is, then determining whether a is admissible requires executing a deconstructor. The deconstructor to be executed for any given pattern is fixed at compile time, using the procedure described in Section 4.3. Executing the deconstructor produces a boolean success value, and one value for each parameter in $\vec{p}$. If the success value is false, a is not admissible. If the success value is true, each value produced by the deconstructor is (recursively) tested for admissibility against its corresponding parameter in $\vec{p}$. The argument a is admissible if all of the values are admissible.

   If a is null and p is a pattern, a is never admissible, and the deconstructor is not executed.

Also, there is an additional exception to the above rules: if the static type of the argument a is neither a subtype nor a supertype of the type of the parameter (either the type of the formal or, in the case of patterns, the type preceding the deconstructor), then a is never admissible. This exception allows programmers to use casting to resolve ambiguities, as mentioned at the end of Section 3.5.1.

4.4.2 Preferred Methods

We now define the preference preorder $\prec_M$ between methods, which is used to determine which of a set of applicable methods shall be invoked. The order is defined in terms of an analogous order $\prec_P$ on method parameters. For two methods m, m' with parameter lists $\vec{p}, \vec{p'}$, m is preferred over m', denoted $m \prec_M m'$, when one of the following conditions holds:

1. m is in a subclass of m', or
2. m and m' are in the same class, have the same number of parameters, and each parameter from $\vec{p}$ is preferred to the corresponding parameter from $\vec{p'}$: $\forall i.p_i \prec_P p'_i$.

The parameter preference relation $\prec_P$ is defined inductively as the smallest preorder satisfying the following:

1. $F[T_1] \prec_P F[T_2]$ whenever $T_1 <: T_2$.
2. $C[v, \_] \prec_P F[T]$ whenever the Java expression `(`T`)` v `==` v evaluates to true.
3. $C[v_1, T_1] \prec_P C[v_2, T_2]$ if the Java expression v_1 `==` v_2 evaluates to true, and $T_1 <: T_2$, where T_1 and T_2 are the types of the literals v_1 and v_2.
4. $P[T_1, n, \vec{p}] \prec_P F[T_2]$ when $T_1 <: T_2$.
5. $P[T_1, n_1, \vec{p_1}] \prec_P P[T_2, n_2, \vec{p_2}]$ when $T_1 <: T_2$, both patterns are associated with the same deconstructor, and $\forall i.{p_1}_i \prec_P {p_2}_i$.

The following lemma is part of ensuring that our notion of parameter preference is a preorder, which is needed in our proof of safety (found in the companion thesis [Ric07]).

LEMMA 4.2. *The parameter preference relation $\prec_P$ is antisymmetric.*

4.4.3 Overall Method Dispatch

To select the method to be invoked for a given call, OOMatch considers all methods in the runtime class of the receiver object and all its superclasses. The method to be invoked for a given call must be applicable for the call, and it must be preferred over all other methods applicable for the call. When exactly one method satisfies these conditions, the method is invoked. Because $\prec_M$ is antisymmetric, it is not possible for more than one method to satisfy the conditions. When no method satisfies the conditions, a runtime error occurs. This can occur if the set of applicable methods is empty (a "no such method" error), or if none of the applicable methods is preferred over all the others (an ambiguity error). In Section 4.6.4, we will present a set of static conditions that guarantee that these runtime errors will not occur.

4.5 Compile-time checks

In this section, we specify properties that the OOMatch compiler checks statically to reduce the number of errors that can occur at runtime. We begin by defining the notion of parameter intersection, which is used in the static checks.

4.5.1 Parameter Intersection

Intuitively, one of the conditions that we would like to hold is that when two methods are both applicable for a call, it will be possible to find a preferred method for the call. For this reason, we define the notion of intersection, a partial function from a pair of parameters to a parameter. We would like intersection to have the following properties:

1. Whenever it is possible for both m_1 and m_2 to be applicable for the same call, the intersections of their corresponding parameters should all be defined, and a method m_3 with those intersections as its parameters should also be applicable for the same call (provided its deconstructors do not return false or null).
2. Whenever the intersection p_3 of two parameters p_1 and p_2 is defined, it should be preferred over both of them: $p_3 \prec_P p_1$ and $p_3 \prec_P p_2$.

Thus, loosely, as long as an OOMatch program contains the intersection of every pair of methods for which intersection is defined, it will not encounter a run-time ambiguity between any pair of methods. We will formalize this property in Section 4.6.4.

We now define a concrete parameter intersection function which we claim satisfies the above properties.

DEFINITION 4.2. *Several cases of the parameter intersection function $\sqcap$ are shown in Table 1. The function is defined to be symmetric; thus, the blank entries in the table correspond to entries opposite the diagonal.*

The intersection of two patterns is the most complicated case. Let $\alpha = P[\theta_\alpha, n_\alpha, \vec{P_\alpha}]$ and $\beta = P[\theta_\beta, n_\beta, \vec{P_\beta}]$. Then the intersection $\alpha \sqcap \beta$ is determined by the following steps:

1. *If α and β correspond to different statically determined deconstructors, their intersection is undefined. Otherwise, proceed to the next step.*
2. *Define θ as follows. If $\theta_1 <: \theta_2$, then $\theta = \theta_1$. If $\theta_2 <: \theta_1$, then $\theta = \theta_2$. If neither of these holds, $\alpha \sqcap \beta$ is undefined. Otherwise, proceed to the next step.*
3. *If $|\vec{P_\alpha}| \neq |\vec{P_\beta}|$, then $\alpha \sqcap \beta$ is undefined. Otherwise, proceed to the next step.*
4. *If for any i, ${P_\alpha}_i \sqcap {P_\beta}_i$ is undefined, then $\alpha \sqcap \beta$ is undefined. Otherwise, proceed to the next step.*
5. *$\alpha \sqcap \beta$ is defined to be $P[\theta, n_\alpha, \vec{P_\alpha} \sqcap \vec{P_\beta}]$, where $\vec{\alpha} \sqcap \vec{\beta}$ is defined as a list of the pairwise intersection of each element in $\vec{\alpha}$ and $\vec{\beta}$.*

4.5.2 Conditions to be checked statically

We now define the conditions under which a class with its set of methods is considered valid or well-formed. Consider a class C, which is valid by the Java rules, and let M_C be the set of methods

$\sqcap$	$\alpha = F[\theta_1]$	$\alpha = C[v_1, \theta_1]$	$\alpha = P[\theta_1, n, \vec{P_\alpha}]$
$\beta = F[\theta_2]$	α if $\theta_1 <: \theta_2$ β if $\theta_2 <: \theta_1$ undefined otherwise		
$\beta = C[v_2, \theta_2]$	β if $(\theta_1)v_1 == v_2$ undefined otherwise	α if $v_1 == v_2$ and $\theta_1 <: \theta_2$ β if $v_1 == v_2$ and $\theta_2 <: \theta_1$ undefined otherwise	
$\beta = P[\theta_2, n_2, \vec{P_\beta}]$	β if $\theta_2 <: \theta_1$ $P[\theta_1, n_2, \vec{P_\beta}]$ if $\theta_1 <: \theta_2$ and $\theta_2 \not<: \theta_1$ undefined otherwise	undefined	see Def 4.2

Table 1. Partial function $\sqcap$ (parameter intersection)

in C. All of the following conditions must hold in order for the class to be accepted by the OOMatch compiler.

CONDITION 4.1. *Unambiguity: For any pair of methods such that neither is preferred to the other and the intersection of their parameter lists is defined, there is some method in C whose parameter list is exactly that intersection. That is, $\forall m_1 = M[\theta_1, n, \vec{\phi_1}, \vec{\theta_{T1}}], m_2 = M[\theta_2, n, \vec{\phi_2}, \vec{\theta_{T2}}] \in M_C$, if $\vec{\phi_1} \sqcap \vec{\phi_2}$ is defined, and $m_1 \not\prec_M m_2$, and $m_2 \not\prec_M m_1$, then $\exists M[\theta_3, n, \vec{\phi_1} \sqcap \vec{\phi_2}, \vec{\theta_{T3}}] \in M_C$.*

CONDITION 4.2. *Valid method calls: For each method call site in the program on a receiver of static type C, there is some method $m = M[T_r, n, \vec{p}, \vec{T_t}]$ implemented in C or its superclasses such that the static types of the argument to the call are subtypes of the* Java *types of $\vec{p}$, as defined by the type function in Section 4.3.*

CONDITION 4.3. *Completeness: This condition is used to prevent no-such-method errors, unless the programmer overrides it by labelling methods* `inc` *(see Section 3.1). Every method m in a class C that contains patterns must be preferred over another method in C or a superclass of C that contains only Java formal parameters, or m must be labelled* `inc`.

In functional languages, ambiguity is resolved by the lexical order of the patterns, and completeness is ensured by statically checking that all cases of any variant types have been handled. With classes, it is not possible to statically check all cases, because the complete set of subclasses of a base class is not known to the compiler. Scala [OAC$^+$06] allows completeness checking by allowing the programmer to fix all subclasses of a class by declaring it `sealed`.

The next two conditions are meant to ensure that if two methods might simultaneously apply, their return types and throws clauses must be compatible. This requirement enables us to know, given a call site, what type of value is returned and what exceptions the method throws. First, we need to formally define what it means for it to be possible for two methods to simultaneously apply.

DEFINITION 4.3. *can-both-apply(m_1, m_2) is a predicate on two methods $m_1 = M[\theta_1, n_1, \vec{\alpha}, \vec{\theta_{T1}}]$ and $m_2 = M[\theta_2, n_2, \vec{\beta}, \vec{\theta_{T2}}]$, in the same class or in a subclass or superclass. It is true if and only if all of the following hold:*

- *$n_1 = n_2$ (Same names.)*
- *$|\vec{\alpha}| = |\vec{\beta}|$ (Same number of parameters.)*
- *$\forall i$, either $\alpha_i \sqcap \beta_i$ is defined, or both of these conditions hold:*
  - *α_i and β_i are both reference types or pattern types, and*
  - *One of α_i and β_i is an interface, or type(α_i) $<:$ type(β_i), or type(β_i) $<:$ type(α_i).*

CONDITION 4.4. *Valid return types: For any pair of methods m_1, m_2 such that can-both-apply(m_1, m_2), their return types must be the same.*

CONDITION 4.5. *Valid "throws" clauses: For any pair of methods m_1, m_2 such that can-both-apply(m_1, m_2), their throw types must be the same, with one exception. If one of the methods is preferred to the other (without loss of generality assume $m_1 \prec m_2$), and m_1 and m_2 both have only regular Java parameters (no patterns, literal values, or "where" clause), then the throw types for m_1 need only be a* subset *of the throw types of m_2. (This exception is important for backward compatibility with Java.)*

CONDITION 4.6. *No duplicate methods: For any two methods $m_1 = M[\theta_1, n, \vec{\alpha}, \vec{\theta_{T1}}]$, $m_2 = M[\theta_2, n, \vec{\beta}, \vec{\theta_{T2}}] \in M_C$, it is not the case that all the parameters are equal; i.e. there is some i such that $\alpha_i \neq \beta_i$. (Unless at least one method has a "where" clause, in which case this condition does not apply.)*

Note that the conditions given here apply only to OOMatch code. There are special rules in place when an OOMatch class extends a regular Java class, which are discussed in the thesis [Ric07].

4.6 Absence of runtime ambiguities

In addition to the conditions above, which are checked by the compiler and must hold in order for an OOMatch program to compile, we define the following optional conditions. If an OOMatch program satisfies these conditions, every call resolves to some method (i.e., no method ambiguity errors can occur).

4.6.1 Undecidable equivalence

An undecidable equivalence is a formalization of the problem mentioned in Section 3.5, when two methods have a corresponding parameter that use different deconstructors that are deconstructing related types. Formally:

DEFINITION 4.4. *undecidable-equivalence is a predicate on pairs of OOMatch parameters. undecidable-equivalence(α, β) is true if and only if $\alpha = P[\theta_1, n, \vec{\phi_1}]$ and $\beta = P[\theta_2, n_2, \vec{\phi_2}]$, where deconstructor$(\alpha)$ $\neq$ deconstructor(β), and either $\theta_1 <: \theta_2$ or $\theta_2 <: \theta_1$.*

DEFINITION 4.5. *undecidable-equivalence-list is a predicate on pairs of lists of parameters. undecidable-equivalence$(\vec{\alpha}, \vec{\beta})$ is true if and only if $|\vec{\alpha}| = |\vec{\beta}|$ and there exists i such that either:*

- *undecidable-equivalence(α_i, β_i) or*

- $\alpha_i = P[\theta_1, n, \vec{\phi}_1]$ *and* $\beta_i = P[\theta_2, n_2, \vec{\phi}_2]$ *and* *deconstructor*(α_i) = *deconstructor*(β_i) *and* *undecidable-equivalence-list*$(\vec{\phi}_1, \vec{\phi}_2)$.

4.6.2 Common descendent

We need to formalize the notion of a pair of parameters where there is a type that is a subtype of both of them; this is one way in which a run-time ambiguity could occur. We define common-descendent to be a predicate on a pair of parameters as follows.

DEFINITION 4.6. *common-descendent(α, β) is defined for a pair of parameters unrelated by subtyping, i.e. when neither type(α) <: type(β) nor type(β) <: type(α). It is true of if and only if the program contains a class θ distinct from α and β such that θ <: type(α) and θ <: type(β).*

Now we define a function used to determine whether there is an instance of common-descendent within the parameters of a pair of methods.

DEFINITION 4.7. *common-descendent-list($\vec{\alpha}, \vec{\beta}$) is true if and only if $|\vec{\alpha}| = |\vec{\beta}|$ and there exists i such that either:*

- *common-descendent(α_i, β_i), or*
- $\alpha_i = P[\theta_\alpha, n_\alpha, \vec{\alpha}']$ *and* $\beta_i = P[\theta_\beta, n_\beta, \vec{\beta}']$ *and* *deconstructor*(α_i) = *deconstructor*(β_i) *and* *common-descendent-list*$(\vec{\alpha}', \vec{\beta}')$.

CLAIM 4.1. *Because Java disallows multiple inheritance, common-descendent can only be true if at least one parameter is an interface.*

4.6.3 Deterministic deconstructors

We need to briefly define the notion of deconstructors being *deterministic*; all deconstructors should be so, though the compiler is not required to check this because it is undecidable. Informally, it means that a deconstructor always returns the same set of values for a given object; i.e. it acts like a function. Formally, a deconstructor is said to be deterministic if it does not modify the heap, and for any object passed to it, it always returns the same values every time it executes.

4.6.4 Claims of safety

Given the above notation and definitions, we can now make the following claim for an OOMatch program that is well-formed, i.e., which has passed the typechecking described above and whose classes are valid.

CLAIM 4.2. *An ambiguity error cannot occur at runtime unless one of the following conditions is true.*

- *common-descendent-list is true for the parameters of some pair of methods with the same name.*
- *There is an undecidable equivalence between the parameter lists of a pair of methods applicable at the same call site.*
- *Some deconstructor is not deterministic.*

5. Use Case

Pattern matching as dispatch can improve code quality to some degree in a wide range of application domains, but it becomes especially useful in an application with a deep class hierarchy where a different operation must be chosen for each class in the hierarchy. This situation occurs in the AST of a compiler. To illustrate this, we have rewritten a part of the Soot Java Optimization Framework [Soo] to use OOMatch. Soot is used to transform Java bytecode, and contains an AST to represent this bytecode. The class we have rewritten uses Visitor traversal and manual pattern matching to check whether a given portion of the AST has a certain form. In Java, this must be done with if-else blocks, instanceof checks, and casts to handle different cases for the sub-trees of the AST node being processed.

In the OOMatch version, the need for visitors (which is the source of a lot of boilerplate code within Soot) has been completely eliminated and replaced by a single method call. The AST processing code has been made significantly more readable as OOMatch patterns, and the amount of explicit casting and run-time type checks has been drastically reduced. Specifically, the OOMatch version contains only 35 uses of `instanceof`, while the Java version contains 121.

Details of this experiment can be found in the companion thesis [Ric07].

6. Other Related Work

While Section 2 gave background work helpful in understanding OOMatch, the following sections give several other pieces of research that are related to OOMatch that are not essential to its understanding.

6.1 Scala

Scala [OAC+06], like OOMatch, is a language that attempts to merge object-oriented and functional programming, roughly starting with Java as a base. It contains a form of pattern matching called *case classes*. A set of case classes is a class hierarchy which allows objects in the hierarchy to be easily matched or deconstructed; there is special syntax to make this convenient. To take the example from [OAC+06]:

```
abstract class Term
case class Num(x : int) extends Term
case class Plus(left: Term, right : Term)
    extends Term
```

`Num` and `Plus` here are each subclasses of, or "cases of", `Term`. `Num`, for example, can now be constructed by passing a single `int` parameter to its constructor. Variables of type `Term` can then be matched against in a special "match" expression, and `Num.x` can be extracted back (deconstructed) when `Num` matches. For example:

```
Term x = ...;
x match {
    case Plus(y, Num(0)) => y
    case Plus(Num(0), y) => y
    case _ => x
}
```

This code is a selection statements that tests whether `x` matches each of the three patterns in turn; if one matches, it executes the subsequent code and then finishes the `match` statement. Generally, the code is simplifying `x` so that if it has the form y + 0 or 0 + y, it is simplified to y.

Case classes are then similar to algebraic types, but more powerful in that they can be used like regular classes.

Scala also has a feature called *extractors*, described in [EOW07], which are similar to OOMatch deconstructors. These allow the addition of "apply" and "unapply" methods to a class, the latter of which allow objects of the class to be decomposed, and their components returned. Such objects can then be matched in a "match" expression, as above, but in a controlled way.

Despite the similarities between Scala's pattern matching and OOMatch, Scala does not use pattern matching for method dispatch, but only a "match" construct that can appear inside a method body. Cases in Scala match expressions are evaluated in the order in which they appear; unlike OOMatch, Scala does not automatically prefer specific patterns over more general ones.

6.2 OCaml

Objective Caml [CMP07] is a language that combines object-oriented and functional styles, in this case by adding classes and objects to a functional language (ML). It contains regular ML pattern matching with a "match" clause, which allows matching of primitives, tuples, records, and union types.

Matching of record types can be seen as being very similar to object matching, as OOMatch allows. Record types in OCaml are simply tuples with names given to each element, where ordering of elements is irrelevant. Matching a record type involves specifying a pattern that can decompose these elements. However, OCaml does not address the more difficult problem of decomposing an object into components that the class writer specifies, or allowing hiding of information as well as pattern matching on that information. OCaml also does not provide multimethods or any other more general form of dispatch on patterns in which precedence is determined automatically by the compiler.

The F# dialect of OCaml has recently been extended with Active Patterns [SNM07]. The definition of an active pattern is a function that takes an object and returns an element of a sum type (i.e. tagged union). The active pattern function can contain arbitrary code, and can therefore deconstruct the object being matched in arbitrary ways, like an OOMatch deconstructor. The match construct matches on the return value of the active pattern function. Active patterns may be either total or partial. In a total pattern, all possible alternatives are specified in a single deconstruction function that returns one alternative; in this case, completeness is checked statically. A partial pattern is more similar to an OOMatch deconstructor in that the matching code for each alternative appears in a separate function, allowing new alternatives to be added. However, completeness of partial patterns is not checked statically.

6.3 TOM

TOM [MRV03] is a language extension that allows decomposing objects into their component parts and matching them with patterns. It takes a multi-language perspective - the extension can be used in Java, C, and Caml. In TOM, one constructs algebraic types, which are entities that have a one-to-one correspondence with a type in the target language (e.g. a Java class). One then provides "functions" on these types to work with them, mapping calls to these functions to code in the base language being used. Then, one can match an algebraic type with a case-like construct (called "%match"), allowing the pattern that matches to be selected and used.

TOM only provides a case-like construct; matching is not used to directly select one of several functions to execute. Its pattern matching, however, works in much the same way as OOMatch's pattern matching, both involving the deconstructing of objects. Further, TOM includes a way to match lists, which is a useful and powerful feature that OOMatch does not (yet) include.

6.4 Views

Views [Wad87], like OOMatch, attempt to unite pattern matching and data abstraction. A view lets one view a regular class type as if it were a type on which pattern matching can be performed. It converts between the view type and the underlying type with "in" and "out" clauses. To take an example from the paper, the following code defines a view for Peano integers, using the built-in integer type as the underlying type:

```
view int ::= Zero | Succ int
    in n         = Zero,      if n = 0
                 = Succ(n-1), if n > 0
    out Zero     = 0
    out (Succ n) = n + 1
```

The "in" clause lets one construct new instances of these special view types, like a Java constructor. The "out" clause gets information out of the view type, or allows pattern matching on it, like OOMatch deconstructors.

Using views, the only way to get information from an object is by making reference to its declarative form – there are no accessor methods like in Java. This may be fine for a functional language, but in Java an object frequently contains information not found in its interface, and there should be a way to get that information back (safely). Also, there is no mention in [Wad87] of the order in which functions with patterns are checked for applicability, or which functions override which; presumably functions appearing first are considered to have priority. OOMatch, in contrast, determines override relationships based on which method is more specific.

6.5 JMatch

JMatch [LM05] shares with OOMatch the attempt to add pattern matching to Java. It allows patterns containing variable declarations to appear in arbitrary expressions, and JMatch attempts to solve for the variables and initialize them with the solved values. It hence allows code very similar to that found in logic programming.

For example:

```
int x + 10 = 0;
```

would cause `x` to receive the value `-10`.

JMatch also allows iteration over a set of values when more than one value satisfies an expression. For example:

```
int[] array;
...
foreach(array[int x] == 0)
{
...
}
```

would iterate through all cases of `x` such that `array[x] == 0`.

JMatch further allows the arguments of method calls to contain patterns, if they are implemented in a special way. This is quite similar to our special constructors that both construct and deconstruct an object. To use an example from the JMatch paper [LM05], a linked list that allows matching (commonly found in functional programming) could be written in JMatch with a special "returns" statement:

```
class List {
    ...
    Object head;
    List rest;
    public List(Object head, List rest)
        returns(head, rest)
    ( this.head = head && this.rest = rest )
}
```

The expression in brackets following the "returns" statement specifies a condition that JMatch uses both in construction and deconstruction of the object. In either case, it finds a set of substitutions that make the boolean expression true. When the special constructor is used simply as a constructor, the values `head` and `tail` are known, and it tries to find a substitution for the values `this.head` and `this.tail` that make the expression true, assigning the resulting values to those fields. The "returns" clause is ignored for construction.

For deconstruction of the object, the instance variables `this.head` and `this.tail` are known, and JMatch calculates what `head` and `tail` must be to cause the condition to be true. The "returns" clause specifies which of these components, that have now been calculated, to return as components of the object for matching. A pattern that matches one of these Lists might look like this:

```
switch(l)
{
    case List(Integer x, List rest): ...
}
```

This code would match a list of at least size 1 where the first element is an Integer. Note that the components of the "returns" clause correspond to the free variables `Integer x` and `List rest` in the pattern.

JMatch's pattern matching is more powerful than that in OOMatch (since it can appear in any expression), but it does not use pattern matching as a form of dispatch, and so does not consider the problem of which patterns are more specific.

6.6 JPred

JPred [Mil04] adds a powerful form of predicate dispatch to Java. It uses a general "when" clause to dispatch on boolean and arithmetic expressions involving the parameters, much like general predicate dispatch. To make it easier to compute the override relationships, JPred restricts the predicates that can appear in a "when" clause to a decidable (though still very powerful) subset, allowing only primitive values, parameter references, subtype queries (allowing for multimethods), field references and built-in operators. The most noteworthy restriction here is that arbitrary method calls cannot appear in "when" clauses. It then uses an external decision procedure – namely, CVC Lite [CVC] – to determine which methods override which.

The original version of JPred disallowed Java interfaces from being matched in order to achieve proof that typechecking can find all ambiguity errors at compile time. Recently this restriction has been dropped (while retaining the type safety) [FM06], though programmers are required to write methods to resolve the potential ambiguities when interfaces are used. A syntactic sugar is provided to make this easier. We cannot take this approach, as it requires more general predicate dispatch than OOMatch has. As we shall see later, we instead allow interfaces to be matched at the cost of a run-time check for some of these ambiguity errors.

6.7 Match0

Pattern matching on objects has been attempted as a Java library API, as opposed to a language extension [Vis06]. The advantage of having a pure Java implementation obviously comes at the cost of increased verbosity from the programmer's perspective; we consider this cost too great and find it worthwhile to provide special syntax for pattern matching.

6.8 Maya

Maya [BH02] is a domain-specific language for specifying rewriting rules on abstract syntax trees. It is intended for building Java language extensions by specifying how to rewrite new syntax into plain Java. Like OOMatch, Maya resolves the ambiguity between multiple patterns that match a given AST node by choosing the most specific one, and giving an error when no pattern is more specific than all others. However, the ambiguity is detected when the pattern matching occurs, rather than ahead of time as in OOMatch.

7. Conclusion

Pattern matching as dispatch has been found to naturally subsume standard polymorphic dispatch and multimethods. Through designing and building a prototype implementation of this feature as an extension to Java, we have explored many issues and nuances that it raises. In particular, we have attempted to find a balance between safety (finding errors as soon as possible) and flexibility (allowing as many programs as possible that make sense), with a bias towards flexibility. Overall, OOMatch provides a balance of power lying between multimethods and general predicate dispatch. In doing so, it uses a simple, intuitive syntax that allows the easy expression of high-level ideas for rule-based systems, and it yields greater abstraction and extensibility.

References

[BH02] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281. ACM Press, 2002.

[BKK$^+$86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-oriented Programming. In *OOPLSA '86: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 17–29, New York, NY, USA, 1986. ACM Press.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular Open Classes and Symmetric Multiple Dispatch for Java. *SIGPLAN Not.*, 35(10):130–145, 2000.

[CMP07] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. Developing applications with objective caml, 2007. Available at http://caml.inria.fr/pub/docs/oreilly-book/ on 5 Feb 2007.

[CVC] CVC Lite. Avaiable at http://www.cs.nyu.edu/acsys/cvcl/.

[D] D Programming Language. Available at http://www.digitalmars.com/d/ on 23 May 2007.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatch: A Unified Theory of Dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, 1998.

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *LNCS*, pages 273–298. Springer, 2007.

[FM06] Christopher Frost and Todd Millstein. Modularly Typesafe Interface Dispatch in JPred. In *FOOL/WOOD '06: International Workshop on Foundations and Developments of Object-Oriented Languages*. ACM Press, 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJSB96] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 2nd edition*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/ on 16 May 2007.

[JAV] Java 2 Platform, Standard Edition, v 1.4.2 API Specification. Available at http://java.sun.com/j2se/1.4.2/docs/api/ on 5 Feb 2007.

[LM05] Jed Liu and Andrew C. Myers. JMatch: Java plus Pattern Matching. Technical Report 2002-1878, Cornell University, 2002, revised 2005.

[Mil04] Todd Millstein. Practical Predicate Dispatch. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 345–364, New York, NY, USA, 2004. ACM Press.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *CC 2003, Compiler Construction: 12th International Conference*, pages 61–76, 2003.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

[OAC$^+$06] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2006.

[POL] Polyglot Extensible Compiler Framework. Available at http://www.cs.cornell.edu/projects/polyglot/ on 16 May 2007.

[Ric07] Adam Richard. OOMatch: Pattern Matching as Dispatch in Java. Master's thesis, University of Waterloo, 2007. Available at http://plg.uwaterloo.ca/~a5richar/oomatch.pdf.

[SNM07] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 29–40, New York, NY, USA, 2007. ACM.

[Soo] Soot: a Java Optimization Framework. Available at http://www.sable.mcgill.ca/soot/ on 23 May 2007.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2005. ACM Press.

[Vis06] Joost Visser. Matching Objects Without Language Extension. *Journal of Object Technology*, 5(8):81–100, Nov-Dec 2006.

[Wad87] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313, New York, NY, USA, 1987. ACM Press.