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 [3]), but which is not as powerful as general predicate dispatch [7]. 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 [12], 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. Of special importance, we ensure that matching can occur while retaining the complete control of class designers 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, and a proof that the conditions specified guarantee run-time safety.

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 and Motivation

Object-oriented programming languages have become the industry hallmark for writing large programs (of millions of lines of code), and with good reason. Building large systems naturally lends itself to breaking down a task into components, and programmers must be provided with a simple interface to conceptualize and work with these components. A class-based type system provides this quite well.

Functional programming languages, on the other hand, have gained a group of devoted followers not only for their beauty, but also for their stability. Because of their declarative nature and lack of side effects, it is considerably more difficult to write buggy software in a functional language (once the software passes the compiler) than in an object-oriented language. As there is a greater trend toward security and closing security holes, as computers are used for more and riskier applications, and as programs get larger, it is becoming increasingly important to prevent bugs from the start.

The strong static typing of languages like Standard ML [12] and Haskell [5] provides this safety quite well. But most functional languages (with notable exceptions such as OCaml [2] and Scala [14]) do not have built-in support for class-based object-oriented programming as found in languages like Java and C++.

Each methodology - functional and OO programming - is useful for a wide class of problems, and embodies the programming style of a large group of people. It is very difficult to say whether one or the other methodology is ideal for all situations, or whether some mixture is ideal. Indeed, if anyone knows the answer to that question, that person should probably create a programming language. Therefore, we believe that for the time being, it is best to provide a language that supports as many styles as possible. The features embodying each style should further be orthogonal with each other, so that different pieces of code written in different styles can be pieced together to form a program.

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 aren't 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 pat-

terns that match objects with certain properties. The patterns are specified in different method headers (as in ML), and the compiler decides on a natural order to check for a matching pattern, i.e. to check which methods override which. This is specified via subtyping, so this feature subsumes polymorphic dispatch and multimethods. Most important to this approach, it should be noted, is that 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 already simulate it themselves using if-else blocks and casting, and its practical value would be lost.

Though OOMatch is being implemented as an extension to Java, it would likely be trivial to adapt the feature to other similar object-oriented languages, such as C++.

2. Background

2.1 Pattern Matching

Pattern matching is a popular feature of the ML family of languages. It lets one create a function with multiple cases, each case of which handles arguments of a particular form. For example, the following code declares a binary tree structure in SML:

```
datatype tree = Leaf of int
    | Node of tree * tree ;
```

With that definition, each case can be matched with multiple function definitions:

```
fun f(Leaf(x)) = ...
|   f(Node(n1, n2)) = ...
;
```

The first version of `f` is called if a leaf node is passed as an argument, and the second is called for non-leaf nodes. The free variables `x`, `n1`, and `n2` can then be referred to in the body of the function, like any parameters. Note that, if multiple functions match (which can't happen in the above example), the first match is the one chosen.

2.2 Primer on Multimethods

Multimethods are a classic example of a powerful form of dispatch. They allow the method chosen to depend on the run-time types of *all* arguments, rather than just the receiver argument. 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)` [9]. 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 behavior. 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 behavior 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 (!(other instanceof C))
            return false;
        C other = (C)otherObject;
        ...
    }
}
```

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

3. Related Work

The idea of dispatch mechanisms more general and powerful than polymorphic dispatch (the dispatch found in Java) is not new. Multimethods were introduced in CommonLoops [1] and added to Java in MultiJava [3]. This notion was generalized and formalized as predicate dispatch [7], in which 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.

Predicate dispatch is by definition the most powerful form of dispatch. However, while it 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 and compute.

3.1 TOM

TOM [13] 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.

3.2 HydroJ

HydroJ [10] introduces a form of dispatch similar to the one given here, but with a completely different goal – that of allowing the function declarations to be changed without changing the calls to those functions, and vice versa. Its application is ubiquitous and distributed computing, in which several small devices communicate with each other, and one wants to improve one component without having to replace (and waste) the old ones. In HydroJ, this is made easier by its rule that if the number of arguments to a function call exceeds the number of parameters in its definition, the excess arguments are ignored. In more mainstream languages, of course, this would be a compile error.

Function parameters in HydroJ can also contain nested patterns on HydroJ's special types ("semi-structured data"). Moreover, in HydroJ a function with more parameters overrides a function of the same name with fewer parameters. Hence, it contains both a form of dispatch and pattern matching to facilitate this dispatch.

This feature is very useful in ubiquitous and distributed computing applications, which the language is targeted for, but is not well-suited for a general-purpose programming language, as is our goal. In particular, the excess argument rule in HydroJ means that if one accidentally adds extra arguments to a call, the program will compile and run, and silently ignore the arguments, likely resulting in bugs. This goes against our goal of safety through a strong type system. Also, though HydroJ's means of dispatch based on matching objects is much like ours, it doesn't provide a means of decomposing a Java type and safely extracting its internals to perform the match.

3.3 JPred

JPred [11] 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 function calls cannot appear in "when" clauses. It then uses an external decision procedure – namely, CVC Lite [4] – to determine which functions override which.

Though its "when" clause is more powerful than OOMatch's pattern matching, we believe there are many situations in which OOMatch is more practically useful. In particular, it is cumbersome to extract the internals of objects in JPred. Perhaps more importantly, doing so requires the data members of objects to be exposed, which violates encapsulation. Further, JPred disallows Java interfaces from being matched in order to achieve proof that typechecking can find all ambiguity errors at compile time. As we shall see later, we choose to relax this restriction at the cost of a run-time check for some of these errors.

3.4 Views

Views [16] were another attempt to unite pattern matching and data abstraction. The "in" clauses of views are similar to Java constructors and "out" clauses are like OOMatch deconstructors. However, 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 [16] of the order 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.

3.5 Scala

Scala [14] is a language that also 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 [14]:

```
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
}
```

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 [6], 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 in OOMatch, Scala does not automatically prefer specific patterns over more general ones.

3.6 OCaml

Objective Caml [2] is another 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 records types can be seen as being very similar to object matching, as OOMatch allows. However, OCaml pattern matching does not match OCaml objects, nor does OCaml provide multimethods or any other more general form of dispatch on patterns in which precedence is determined automatically by the compiler.

4. Using OOMatch - Informal Description

4.1 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 { ... }
```

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 parameter 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 the way these patterns are written, and which ones are allowed, is so far magic to the reader – the extra information the classes `Plus` and `NumConst` must provide in order to allow themselves to participate in pattern matching is described shortly.

The key point to notice in the above example is that the third method *overrides* the first, since its pattern is *more specific than* the original's (because `Binop` extends `Expr`), and the second 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. Note that the patterns themselves can be named or unnamed; the Plus match is unnamed, while the Binop 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 any arbitrary depth. The most specific match is

always chosen first. So, for example, we could add another function with signature

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

where `IntConst` extends `NumConst`, and this new function would override the third one, because the pattern type is the same but the subpatterns are more specific.

An interesting exercise is to think of what would be necessary to rewrite the above code in pure Java. While it's noteworthy that our version is probably more compact and understandable, a more important property of the OOMatch version is that each method represents a separate rule. This one-to-one mapping of rules to methods is not as feasible in Java. One Java approach would be to have one method do the dispatch to determine which "rule" is applicable - it could be called `optimize` in this case - and have other methods of different names for each rule (either `optimize1`, `optimize2`, and `optimize3`, or perhaps `optimizePlusZero` and `optimizeFold`). There are many situations where this approach is less than elegant.

Note that OOMatch introduces the potential for new kinds of compile errors. In fact, the above code contains such an instance, as a careful reader might have noticed. If `optimize` is passed an expression like `1 + 0`, the second and third methods will 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 doesn't, 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 can 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 called an *incomplete error*. It occurs when there is a method that is not general enough to be called directly from a call site (we shall see later how "not general enough" is defined), and which appears alone (i.e., it doesn't override any other method), so that it has no way of being called. For example, if the following lone function appeared:

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

an incomplete error would result, because the case `NumConst(1)` (among others) is not handled.

4.2 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, programmers can add access specifiers to constructor parameters:

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

The `public` specifier on parameters does 4 things:

- Declares the variable to be a public instance variable of the class
- Declares a parameter to the constructor
- Assigns the argument passed to the constructor to the instance variable
- Allows the object to be deconstructed in a pattern that corresponds to the way it was constructed

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. Note that access specifiers other than "public" are allowed to restrict access to the variable in the class; however, the object can still be deconstructed as long as it has a constructor whose parameters have some access specifier.

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. Indeed, the above definition of the Binop class using the `public` specifier is merely syntactic sugar for the following form.

```
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;
    }
    ...
}
```

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. 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, classes can restrict access to their instance variables however they see fit.

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 solely a result of the limitations of Java. 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 function which takes no parameters and returns a tuple could then be used as a deconstructor. This approach was taken by Scala's extractors [6], for example.

In general, method headers in OOMatch can contain regular formal parameters, or patterns. Patterns can contain literal primitive values (including string literals). One cannot, then, match on a specific object value of class type, unless they provide a deconstructor for the object and specify the value as a pattern. In other words, one cannot specify a "new" expression in a parameter to match against. Also note that literals can appear outside of patterns, in place of regular parameters. For example, the following pair of functions 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. However, 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))
{ ... }
```

This rule could be removed relatively easily in a language with a construct to include a file and bring all symbols defined therein – including deconstructors – into the current namespace, but Java's `import` statement does not allow this. The alternative would be to search all imported classes for possible deconstructor definitions, but this be too expensive.

From the point of view of the OOMatch compiler, referring to a deconstructor as `X.Y` is the "default" 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*. Hence, the expression `Plus(Expr e, NumConst(0))` seen earlier could be short for `Plus.Binop(Expr e, NumConst(0))`, if the class `Plus` does not have a deconstructor of its own.

4.3 No "throws" Clause for a Deconstructor

Deconstructors may not be declared to throw any exceptions. The advantage of a `throws` clause is that it forces clients to catch the exceptions declared in the corresponding method. But the method headers that use the deconstructors obviously cannot themselves catch these exceptions. Another possible rule would be to force method headers to declare a "throws" clause containing any exceptions that the deconstructors it invokes throw. The callers of such methods would then be required to catch these exceptions. However, if this were done, then all methods in an overriding group would also be required to throw the exceptions, even if they do not use the deconstructors themselves. This could get complex and confusing. More importantly, we do not see any obvious use for this feature; if one is discovered, it could become future work.

4.4 Deconstructor Return Value

To further increase the power of deconstructor bodies, they may return a boolean value; `true` means the match succeeded, `false` means it failed. This allows even a deconstructor whose pattern is matched to disallow the match under certain arbitrary conditions, such as the state of the object. This return statement is optional.

4.5 Order of Deconstructors

The order deconstructors are called (when determining which method applies to a method call) is left unspecified. This choice was made to free implementations to do optimizations or maximize other metrics that may require certain implementations of the dispatch algorithm (described later). Further, implementations may choose not to run the deconstructor for a given pattern, as long as the required semantics are preserved. However, we do make the requirement that deconstructors are run at most once to determine the method to call. In particular, the rule is that for a given

method call, a deconstructor referenced in a method of the same name as the call may not be run more than once for each reference to it.

Because of the functional nature of the context in which deconstructors are intended to be used, 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 behavior was deemed more desirable than explicitly-defined behavior, because it increases the potential for optimizations.

4.6 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 code:

```
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 type; syntax to allow this could be a possible future addition. Otherwise, `null` is doing nothing special here; since `null` is a subtype of all other types, it overrides all methods with a single parameter of class type, as expected.

Another trickier issue with `null` is in matching patterns. Given the `Binop` deconstructor from Section 4.2, one might expect the following lone function to present no problems, as it handles every `Binop` object:

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

But unlike a function 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. Unlike ML and similar languages, which disallow the above code and force a function containing `null` to be provided, OOMatch permits the above and assumes the user is never going to pass `null` to `f`. This allows patterns to be used for no other reason than to specify non-null parameters, which is sometimes useful. Besides, many (though not all) bugs caused by passing `null` to a pattern parameter could be caught statically, with a program analysis.

Another way around this problem that was considered is to force every method using a pattern to override a regular Java method; this way, the Java method could always be called as a last resort. This was decided against for the simple reason that one useful way to use patterns is not for override purposes, but merely to extract the internals of an object simply and concisely. In the above example, the user might not care to give a name to the single `Binop` parameter, because she only cares about the two `Expr`s contained within it, and by using patterns is avoiding calls to `Binop`'s "getter" methods within the body of the function. It would be tedious to force programmers to write an extra function that throws an exception for `null` in these cases, especially since the runtime system generates an exception for `null` anyway.

4.7 Undecidable Errors

Though the compiler can catch many of the new possible errors, 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 relatively rare cases described here occur. We now describe the three types of errors, in addition to the "null" error mentioned in Section 4.6 and the possibility of deconstructors returning false, that might cause a run-time exception.

4.7.1 Interfaces

The first potential error is caused by multiple inheritance, which is partially allowed in Java by implementing multiple interfaces. Consider the following trivial pair of functions:

```
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 will not know which version to call. It is not possible to tell whether such a `C` exists at compile time; not only does Java's modular typechecking preclude knowledge of the class hierarchy, but dynamic class loading means that knowing what classes will be present at the time of a call to `f` is undecidable.

A simple test showed that JPred [11] avoided this issue by disallowing interfaces as multimethod parameters. We found this approach too restrictive; programmers expect to be able to use interface parameters the same way they use class parameters. Further, if this feature were migrated to a language with multiple inheritance, the problem would return.

4.7.2 Different deconstructors

The next type of ambiguity can occur when there is a pair of functions in a group, and a corresponding parameter is referring to a different deconstructor in each function. For example, let's take the 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(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's true, and the other when it's 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've 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 functions will 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 can occur if this turns out not to be so.

Note that this problem can appear in mischievous ways. For example, if a subclass defines a deconstructor of the same name as a superclass (not something that is normally useful), the two deconstructors are considered completely separate, and override relationships that may have been assumed to be present may in fact not be present. For example, consider this code:

```
class Point {
    ...
    deconstructor Point(int x, int y) { ... }
}

class ScreenCoordinate extends Point {
    ...
    deconstructor Point(int x, int y) { ... }
}
```

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

One might expect the second version of `f` to override the first, because they both have the same pattern, but the second method only admits objects of type `ScreenCoordinate`. However, because of the deconstructor definition in `ScreenCoordinate`, it is not so; the compiler can say nothing, in general, about which deconstructor is more specific than which, even if the user intended the one in `ScreenCoordinate` to be more specific. Hence, there is no overriding taking place here, nor is there a compile error. Of course, compiler implementations can and should give a warning in this situation.

4.7.3 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 functions which use the class `Point` described above:

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

It may appear that these functions 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, 2 zeroes are returned, and on the second invocation, two ones are returned, which makes both functions 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.

This kind of non-deterministic behavior is, of course, not very useful in a pattern matching context, and is hence relatively easy to avoid; on the other hand, such problems 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.

4.8 Cross-class Ambiguities

An issue that arises when studying multimethods is whether Java's subclass overriding should always take precedence over parameter overriding. The alternative is to give an ambiguity error. For example, consider this code:

```
class Shape {
    intersection(Shape s);
    intersection(Circle(0, 0));
}

class Square extends Shape {
    intersection(Shape s);
}
```

What happens if

```
Square.intersection(new Circle(0, 0))
```

is called? Should `Square.intersection` be ambiguous with `Shape.intersection(Circle(0, 0))`? In OOMatch, we decided to resolve the ambiguity in favor of `Square.intersection` – i.e., to make Java overriding always take precedence over OOMatch overriding. The original thought was that preventing the above code as an ambiguity (which may make perfect sense to the programmer who wrote it) was unnecessarily restrictive.

In retrospect, this choice was probably a mistake. A better, and safer, approach would be to allow the programmer to attach annotations specifying which method should take precedence. Besides safety, this approach has the advantage that it very often happens that a class writer wants to impose a policy on all subclasses; perhaps the writer of Shape wants intersection to always do a specific thing when called with `Circle(0, 0)`. In the current implementation, this is not possible.

This feature is left to future work.

4.9 Backwards Compatibility

The syntax of Java is, for the most part, a subset of OOMatch; that is, most Java code will compile as an OOMatch program. However, there are two exceptions that may cause incompatibilities.

First, there are cases where code that is valid as Java code generates an ambiguity error when compiled with OOMatch. For example, suppose `A extends B` and we have this code:

```
class C {
    void f(A a, A b) { ... }
    ...
}
class D extends C {
    void f(A a, B b) { ... }
    void f(B b, A a) { ... }
}
```

This is fine in Java; all three methods are overloaded. But in OOMatch, the two versions of `D.f` override the version in `C`, but are ambiguous with each other. An error must be given here, because if `C.f` is called statically, but is passed a pair of `B`s and a receiver argument of type `D`, there will be an ambiguity.

Second, the meaning of a Java program might be slightly different when treated as an OOMatch program. In particular, methods which are only overloaded in Java might become overridden when treated as OOMatch code. For example, suppose `B` extends `A` and we have the following Java code:

```
class C {
    void f(A a) {...}
    void f(B b) {...}
    void g() {
        A a = new B();
        f(a);
    }
}
```

if the class is compiled as a Java class, the call to `f` will invoke the first version of `f`, despite the fact that the argument's "real" type is `B`, . If compiled as an OOMatch class, however, the two versions of `f` become multimethods, and the second version is invoked. Though the Java behavior may seem stranger, there may be legacy code that depends on it, and whose behavior should not be changed; that it is changed is a point against OOMatch. Nevertheless, this disadvantage was deemed a worthwhile price to pay to avoid the need for specific syntax for multimethod behavior (such as the "@" symbol from MultiJava [3]). Special syntax would make the feature more cumbersome and confusing to learn, which, we felt, is worse than violating backwards-compatibility.

Further, it should be noted that OOMatch code is interoperable with Java code. It is entirely possible to compile new code with the OOMatch compiler, and use the resulting .class file with legacy code that has been compiled with the Java compiler. When assurance is needed that legacy code retain its exact behavior, or when Java code that will not compile with OOMatch cannot be updated, part of the code can be compiled with the Java compiler.

5. Formal Specification

5.1 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 [8]. 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)`.

5.2 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 throwing types $\vec{T_t}$.

We explicitly define a subtyping relation that corresponds to the assignability rules defined in the Java Language Specification [8].

DEFINITION 5.1. *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. *$\texttt{byte} <: \texttt{short} <: \texttt{int} <: \texttt{long} <: \texttt{float} <: \texttt{double}$, and $\texttt{char} <: \texttt{int}$.*
4. *For array types, $A\texttt{[]} <: B\texttt{[]}$ if $A <: B$.*
5. *$T <: \texttt{Object}$ for all class, interface, and array types T.*
6. *$T[] <: \texttt{Cloneable}$ and $T[] <: \texttt{java.io.Serializable}$ for all array types $T[]$.*

LEMMA 5.1. *Subtyping is a partial order.*

Proof We have transitivity and reflexivity from the definition; we only need to show antisymmetry, i.e. no cycles.

The subtyping cases can be divided into primitives, arrays, and other reference types. Primitive types are unrelated to any other types, and there is no cycle in the relations given in case 3. Array types are not the supertype of any other reference type, so there can be no cycles between them and other reference types. From case 4, there can be no cycles among array types unless there are cycles among reference types. Null is not the supertype of any other type, so it cannot participate in a cycle; the only thing left is classes and interfaces. These cannot be defined circularly, as stated in the Java specification, section 8.1.3. [8] ∎

5.3 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

- 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.

5.4 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.

5.4.1 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: $\forall i.$*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 [8, 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 which that was described in Section 5.3. 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 admissable.

5.4.2 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 $\vec{p}, \vec{p'}$, $m \prec_M m'$ 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. $F[T_1] \prec_P P[T_2, n, \vec{p}]$ when $T_1 <: T_2$ and it is not the case that $T_2 <: T_1$.
6. $P[T_1, n_1, \vec{p_1}] \prec_P P[T_2, n_2, \vec{p_2}]$ when $T_1 <: T_2$, both patterns are associated with the same deconstructor, and $\forall i. p_{1i} \prec_P p_{2i}$.

LEMMA 5.2. *The parameter preference relation $\prec_P$ is antisymmetric.*

Proof

We need to prove that $a \prec_P b$ and $b \prec_P a$ implies $a = b$. We show it by structural induction on the parameters. There are six cases to consider.

1. $a = F[T_1], b = F[T_2]$. Then $T_1 <: T_2$ and $T_2 <: T_1$. Since there are no cycles in subtyping, T_1 and T_2 are the same; so, by definition, $a = b$.
2. $a = C[v, T_1]$ and $b = F[T_2]$. The condition is always false, because $F[T_2] \not\prec_P C[v, T_1]$ for any formal and constant parameters, so the implication is true by default.
3. $a = C[v_1, T_1]$ and $b = C[v_2, T_2]$. This means that $T_1 = T_2$, since $T_1 <: T_2$ and $T_2 <: T_1$. And we also know $v_1 == v_2$ by the definition of $\prec_P$. So, by definition, $a = b$.
4. $a = P[T_1, n, \vec{p}], b = F[T_2]$. If $a \prec_P b$, $T_1 <: T_2$. Since $b \prec_P a$, it isn't the case that $T_1 <: T_2$. This is a contradiction, so the implication is true by default.

5. $a = F[T_1], b = P[T_2, n, \vec{p}_2]$. WLOG, this is the same case as the previous one.
6. $a = P[T_1, n_1, \vec{p}_1], b = P[T_2, n_2, \vec{p}_2]$. To have $a \prec_P b$ and $b \prec_P a$, $T_1 = T_2$ for the reasons given above. Since the deconstructors are the same, the names n_1 and n_2 must be equal. $\vec{p_1} = \vec{p_2}$ by induction. So, it follows by definition that $a = b$.

∎

5.4.3 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 5.6.4, we will present a set of static conditions that guarantee that these runtime errors will not occur.

5.5 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 notions of parameter intersection and always-matches, which are used in the static checks.

5.5.1 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 5.6.4.

We now define a concrete parameter intersection function which we claim satisfies the above properties. We will prove the claim in Section 5.6.4.

DEFINITION 5.2. *Several cases of the parameter intersection function 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 $P[\theta, n_\alpha, \vec{P_\alpha} \sqcap \vec{P_\beta}]$.*

5.5.2 Always-matches

A second informal condition that we would like to enforce is that given a call site, for any actual runtime arguments satisfying the declared types at the call site, some method will be applicable (again, provided that deconstructors do not return false or null, and null is not passed to them).

We define the predicate *always-matches*$(\vec{T_s}, \vec{P})$ on a list of static types (i.e. the static types of the call site arguments) and a list of OOMatch parameters (i.e. the parameters of a method) as follows:

DEFINITION 5.3. *always-matches$(\vec{T_s}, \vec{P})$ is true if both of the following conditions hold:*

1. *$|\vec{T_s}| = |\vec{P}|$, and*
2. *for every i, either*
   (a) *$P_i = F[T]$ and $T_{si} <: T$, or*
   (b) *$P_i = P[T, n, \vec{P'}]$ and $T_{si} <: T$ and, letting $D[n, \vec{T'_s}]$ be the deconstructor bound to P_i, always-matches$(\vec{T'_s}, \vec{P'})$.*

LEMMA 5.3. *Let $m = M[T_r, n, \vec{p}, \vec{T_t}]$ be a method, and $\vec{a}$ the actual arguments of a method call with static types $\vec{T_s}$. If always-matches$(\vec{T_s}, \vec{p})$, and no deconstructor returns false or null, and none of the arguments a_i are null, then applicable$(m, n, \vec{a})$.*

Proof

The predicate *applicable* from section 5.4.1 has 3 conditions. The first holds by our assumption that the method call is to n. The second holds because *always-matches*$(\vec{T_s}, \vec{p}) \Rightarrow |\vec{T_s}| = |\vec{p}|$. The third condition requires that each of a_i is

$\sqcap$	$\alpha = F[\theta_1]$	$\alpha = C[v_1, \theta_1]$	$\alpha = P[\theta_1, n, \vec{P_\alpha}]$
$\beta = F[\theta_2]$	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 5.2

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

admissible to p_i. This is shown next, by contradiction and structural induction.

There are three cases for p_i:

1. $p_i = C[\_, \_]$. In this case, *always-matches*$(\vec{T_s}, \vec{p})$ must have been false, because there is no case for literal parameters in the definition of *always-matches*. Contradiction.
2. $p_i = F[T]$. We need to look at case 1 of *admissible*. Since *always-matches*$(\vec{T_s}, \vec{p})$, $T_{si} <: T$. We need to show that this implies that T_{si} can be converted by method invocation conversion to T.

   There are three cases for T_{si}: primitive, array, or other reference type. According to [8, Section 5.3], widening primitive conversions or widening reference conversions (or identity conversions) must be allowed for method invocation conversion to be doable. For primitives, only case 3 of $<:$ applies; the subtyping relations given there correspond exactly to the widening primitive conversions in [8, Section 5.1.2]. For arrays and other reference types, cases 1, 4, 5, and 6 apply, and correspond exactly to the widening reference conversions in [8, Section 5.1.4]. Hence, *admissible* holds in this case.
3. $p_i = P[T, n, \vec{p'}]$. We need to look at case 3 of *admissible*. Since *always-matches*$(\vec{T_s}, \vec{p})$, $T_{si} <: T$, so the first part of the case passes. The deconstructor is then executed, and because we assume it returns true, the second part of the conditions also pass, and we get a new set of values with types $\vec{T_s'}$. We assume that a_i is not null, so that can't prevent *admissible* from holding. From the definition of *always-matches*, *always-matches*$(\vec{T_s'}, \vec{p'})$. Furthermore, we assume that none of the values returned from the deconstructor are null. Therefore, by the inductive assumption, admissibility holds.

∎

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

CONDITION 5.1. *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 $m_1 \not<_M m_2$, and $m_2 \not<_M m_1$, then $\exists M[\theta_3, n, \vec{\phi}_1 \sqcap \vec{\phi}_2, \vec{\theta_{T3}}] \in M_C$.*

CONDITION 5.2. *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 always-matches$(\vec{T_s}, \vec{p})$, where $\vec{T_s}$ are the static types of the arguments at the call site. Moreover, one of the methods satisfying this condition is preferred over all methods satisfying the condition.*

CONDITION 5.3. *Completeness: Loosely, for every method in the program, it is possible to construct a call site that could call the method. Formally, for every method $m = M[T_r, n, \vec{p}, \vec{T_t}]$ declared in a class C in the program, if we were to add to the program the call site $o.n(\vec{a})$, where the static type of o is C and the static types of $\vec{a}$ are type$(\vec{p})$, then the previous condition would hold for this added call site.*

CONDITION 5.4. *Valid return types: For any pair of related methods, the return type must be the same. Specifically, 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 C or a superclass of C such that the intersection of their parameter lists is defined, the return types must be the same, i.e. $\theta_1 = \theta_2$.*

CONDITION 5.5. *Valid "throws" clauses: For any pair of related methods, the throws clauses must be the same. Specifically, 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 C or a superclass of C such that the intersection of their parameter lists is defined, all the types in $\vec{\theta_{T1}}$ are also in $\vec{\theta_{T2}}$.*

CONDITION 5.6. *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 $\alpha_i \neq \beta_i$.*

5.6 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 will resolve to some method (i.e. no method ambiguity errors can occur).

5.6.1 Undecidable equivalence

An undecidable equivalence is a formalization of the problem mentioned in Section 4.7, when two methods have a corresponding parameter that use different deconstructors that are deconstructing related types. Formally:

DEFINITION 5.4. *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 5.5. *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$(\vec{\phi}_1, \vec{\phi}_2)$.*

5.6.2 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 5.6. *common-descendent(α, β) is true if and only if the program contains a class θ distinct from α and β such that $\theta <:$ type(α) and $\theta <:$ 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 5.7. *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 5.1. *Because Java disallows multiple inheritance, common-descendent can only be true for a pair of interfaces.*

5.6.3 Deterministic deconstructors

We need to briefly define the notion of deconstructors being *deterministic*; all deconstructors should be so, though the compiler isn't required to check this because it's 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.

5.6.4 Claims of safety

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

CLAIM 5.2. *A no-such-method error cannot occur at runtime unless one of the following occurs.*

- *A deconstructor returns false.*
- *A deconstructor returns `null` in one of its "out" parameters.*
- *`null` is passed to a pattern parameter.*

Proof This claim follows from Condition 5.2 and Lemma 5.3. Since for each call site, there is a method for which *always-matches$(\vec{T}_s, \vec{p})$* is true, where $\vec{T}_s$ are the static types of the arguments of the call, and $\vec{p}$ are the parameters of the method. Lemma 5.3 says that this method is applicable. ∎

CLAIM 5.3. *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 to the same call site.*
- *Some deconstructor is not deterministic.*

Proof

Let $o.n(\vec{r})$ be any method call site. Let A be the set of methods applicable for the call. We assume that at least one method is applicable (otherwise, a no-such-method error occurs). Let B be any nonempty subset of A. We will show by induction on $|B|$ that for every set B, there is a method in A that is preferred over all methods in B.

Base case: $|B| = 1$. By reflexivity of the preference relation, the single method in B is preferred over all methods in B.

Inductive case: Suppose that for every subset C of size $|C| = k$ of A, there is a method in A that is preferred over every method in C.

Let B be any subset of A of size $|B| = k + 1$. Select any method m_1 from B. Since the set $B \setminus \{m_1\}$ is of size k, there is a method m_2 in A that is preferred over all methods in $B \setminus \{m_1\}$. We have three cases to consider:

1. $m_1 \prec_M m_2$. Then m_1 is preferred over all methods in B, since $\prec_M$ is transitive.
2. $m_2 \prec_M m_1$. Then m_2 is the unique method preferred over all methods in B.
3. $m_1 \not\prec_M m_2$ and $m_2 \not\prec_M m_1$. Then, letting $\vec{p_1}$ and $\vec{p_2}$ be the parameter lists of m_1 and m_2, the intersection $\vec{p_1} \sqcap \vec{p_2}$ is defined, by Lemma 5.4 which will be proven below. By the definition of the $\prec_M$ relation, m_1 and m_2 must be in the same class (otherwise, one would be preferred over the other). By Condition 5.5.3, there is a method m_3 implemented in the same class as m_1 and m_2 whose parameters are $\vec{p_1} \sqcap \vec{p_2}$. An additional consequence of Lemma 5.4 is that m_3 is applicable, and therefore in A. By Lemma 5.5 (given below), m_3 is preferred over m_1 and m_2. By transitivity of the preference relation, m_3 is preferred over all methods in B.

By induction, for every subset B of A, including A itself, there is a method in A preferred over all methods in B. ∎

LEMMA 5.4. *Let $\vec{p_1}$ and $\vec{p_2}$ be a pair of parameter lists with $|\vec{p_1}| = |\vec{p_2}|$. Additionally, suppose that common-descendent-list$(\vec{p_1}, \vec{p_2})$ and undecidable-equivalence-list$(\vec{p_1}, \vec{p_2})$ are both false. Also, suppose that all deconstructors associated with all patterns in $\vec{p_1}$ and $\vec{p_2}$ and all of their subpatterns are deterministic. Finally, suppose that there is a list of actual Java values $\vec{r}$ such that each value is admissible for the corresponding parameter in both p_1 and p_2. Then the intersection $\vec{p_1} \sqcap \vec{p_2}$ is defined, and the same values $\vec{r}$ are admissible for the intersected parameters $\vec{p_1} \sqcap \vec{p_2}$.*

Proof

Let α be any parameter of $\vec{p_1}$ and β be the corresponding parameter of $\vec{p_2}$. Let r be the actual argument admissible for both α and β. We will show by structural induction on the forms of α and β that $\alpha \sqcap \beta$ is always defined.

- Suppose $\alpha = C[v_1, \theta_1]$ and $\beta = C[v_2, \theta_2]$. Since both methods are applicable for the call, v_1 `==` r and v_2 `==` r. There are 3 cases for r.
  1. r is a non-null `String` literal. Since `String` literals only `==` other `String` literals, v_1 and v_2 must be the same `String` literal, so v_1`==`v_2. Therefore, $\alpha \sqcap \beta = \alpha = \beta$.
  2. r is `null`. Since `null` is only `==` to `null` (see section of [8, Section 15.21.3]), v_1 and v_2 are both `null`. Therefore v_1`==`v_2, so $\alpha \sqcap \beta = \alpha = \beta$.
  3. r is of a primitive numeric type (including `char`). Now, for numeric types, `==` is an equivalence, because [8, Section 15.21.1] states "The value produced by the == operator is true if the value of the left-hand operand is equal to the value of the right-hand operand; otherwise, the result is false.", with the exception of NaN, which is not equal to itself. But NaN cannot appear as a constant parameter, because there is no floating point constant that can represent it (see [8, Section 3.10.2]). This means that v_1 `==` v_2. And it cannot happen that two numeric values are equal unless the type of one is a subtype of another, or one is `short` or `byte`. But there is no constant parameter of type `short` or `byte`, because all integer literals have type `int` (see [8, Section 3.10.1]); therefore $\theta_1 <: \theta_2$ or $\theta_2 <: \theta_1$. Therefore, $\alpha \sqcap \beta$ is defined to be one of α or β.
- Suppose α is a literal $C[v, \theta_1]$ and β is a formal $F[\theta_2]$. Letting T_d be the run-time type of r, we know that $T_d <: \theta_2$ and $r == v$. Now, there are three cases for the type θ_1 of the constant parameter v.
  1. v is a non-null `String` literal. In this case, the only values for r such that $r == v$ is another `String` literal; in other words, θ_2 = `String`. But `(String)`v`==`v for any `String` literal v. Therefore $C[v, \theta_1] \sqcap F[\theta_2]$ is defined to be $C[v, \theta_1]$.
  2. v is `null`, and θ_1 = NullType. But the only value that's equal to `null` is `null` (see [8, Section 15.21.3]), and (NullType)`null == null`. Therefore, $C[v, \theta_1] \sqcap F[\theta_2]$ is defined to be $C[v, \theta_1]$.
  3. v and r are numeric types. For numeric types, `==` is an equivalence (see above). Since testing equality involves binary numeric promotion ([8, Section 5.6.2]), it is also the case that == holds if r and v are treated as having the type that they're both widened to. Hence, there are 3 further sub-cases.
     - $\theta_1 <: \theta_2$.

       Then $(\theta_2)v == v$ because, by binary numeric promotion this is the same as $(\theta_2)v == (\theta_2)v$, which is true.
     - $\theta_2 <: \theta_1$.

       Then we know that $(\theta_1)r$`==`$(\theta_1)v$, since $r == v$ and by binary numeric promotion.

       Therefore, $(\theta_2)(\theta_1)r$`==`$(\theta_2)(\theta_1)v$, because the values being narrowed are equal, so will still be equal after narrowing.

       Therefore, $(\theta_2)(\theta_1)r$`==`$(\theta_2)v$, since v has type θ_1.

       Therefore, r`==`$(\theta_2)v$, since this means that same as the previous form because of binary numeric promotion.

       Therefore, by transitivity, v`==`$(\theta_2)v$.

- θ_2 and θ_1 are unrelated, and have the common supertype `int`.

  Then binary numeric promotion says that $r == v$ is the same as $(int)r == (int)v$.

  Therefore, $(\theta_2)(int)r == (\theta_2)(int)v$, because the values being narrowed are equal, so will still be equal after narrowing.

  Therefore, $r == (\theta_2)(int)v$, since this means the same as the previous form because of binary numeric promotion.

  Therefore, $r == (\theta_2)v$, for the same reason. (Values smaller than `int` are always widened to `int`).

  Therefore, by transitivity, v`==`$(\theta_2)v$.

So in each case, $(\theta_2)v == v$. Therefore, $C[v, \theta_1] \sqcap F[\theta_2]$ is defined to be $C[v, \theta_1]$.

- It is not possible for α to be a literal and β to be a pattern, because there are no values admissible to both a literal and a pattern. The only values admissible to a literal are values of primitive types, `null`, or values of type `String`. Patterns never match values of primitive type or `null`. Since the class `String` is final, it is not possible to add a deconstructor to it, so it also cannot be matched by a pattern.
- Suppose α and β are both formals $F[\theta_1]$ and $F[\theta_2]$. Let T_d be the run-time type of r. Since $T_d <: \theta_1$ and $T_d <: \theta_2$, the lack of a common descendent of α and β implies that $T_d = \theta_1$ or $T_d = \theta_2$. Without loss of generality, assume the former. Then $\theta_1 <: \theta_2$, so $\alpha \sqcap \beta$ is defined to be α.
- Suppose α is a pattern $P[\theta_1, n, \vec{\phi}]$ and β is a formal $F[\theta_2]$. By the same reasoning as in the previous case, either $\theta_1 <: \theta_2$, or $\theta_2 <: \theta_1$. In both of these cases, the intersection is defined, specifically as either α or $P[\theta_2, n, \vec{\phi}]$. Now, $P[\theta_1, n, \vec{\phi}]$ and $P[\theta_2, n, \vec{\phi}]$ must be associated with the same deconstructor; otherwise, *undecidable-equivalence*$(P[\theta_1, n, \vec{\phi}], P[\theta_2, n, \vec{\phi}])$ would be true. Since r is admissible to both α and β, and since deconstructors are deterministic, it is also admissible to $P[\theta_2, n, \vec{\phi}]$.
- Suppose $\alpha = P[\theta_\alpha, n_\alpha, \vec{p_\alpha}]$ and $\beta = P[\theta_\beta, n_\beta, \vec{p_\beta}]$ are both patterns. By the same reasoning as in the previous two cases, $\theta_\alpha <: \theta_\beta$ or $\theta_\beta <: \theta_\alpha$. Without loss of generality, assume the former. Now α and β must be associated with the same deconstructor; otherwise, *undecidable-equivalence*(α, β) would be true. Since α and β are associated with the same deconstructor, $|\vec{p_\alpha}| = |\vec{p_\beta}|$. Since the deconstructor is deterministic, evaluating it returns the same values for both patterns. Therefore, Lemma 5.4 can recursively be applied to $\vec{p_\alpha}$ and $\vec{p_\beta}$. Therefore, $\alpha \sqcap \beta$ is defined, specifically to $\gamma = P[\theta_\alpha, n_\alpha, \vec{p_\alpha} \sqcap \vec{p_\beta}]$. Since deconstructors are deterministic and r is admissible to both α and β, it is also admissible to their intersection γ.

We have shown that for all possible forms of α and β, $\alpha \sqcap \beta$ is defined and r is admissible to it. ∎

LEMMA 5.5. *For a pair of parameters a and b whose intersection is defined, if undecidable-equivalence(a, b) is false, then $c \prec_P a$ and $c \prec_P b$, where $c = a \sqcap b$.*

Proof There are six cases to consider.

1. $a = F[\theta_1], b = F[\theta_2]$. If $\theta_1 <: \theta_2$ then $c \prec_P b$ holds from case 1 of its definition, and $c \prec_P a$ is true by reflexivity. WLOG, the case where $\theta_2 <: \theta_1$ is the same.
2. $a = F[\theta_1], b = C[v_2, \theta_2]$. The only possibility for their intersection to not be $\bot$ is for it to be b. $b \prec a$ from case 2 of the definition of $\prec_P$, which is the same as the conditions of $\sqcap$, and $b \prec_P b$ by reflexivity.
3. $a = C[v_1, \theta_1], b = C[v_2, \theta_2]$. If $\theta_1 <: \theta_2$ and $v_1 == v_2$, then the intersection is a. $a \prec_P b$ by rule 3 of $\prec_P$, and $a \prec_P a$ by reflexivity. WLOG, the case where $\theta_2 <: \theta_1$ is the same.
4. $a = F[\theta_1], b = P[\theta_2, n_2, \vec{\beta}]$. If $\theta_2 <: \theta_1$ and $a \sqcap b = b$, then $b \prec_P a$ by case 4 of $\prec_P$, and $b \prec_P b$ by reflexivity.

   Suppose $\theta_1 < \theta_2$ and $a \sqcap b = P[\theta_1, n_2, \beta] = c$. Then we have $c \prec_P a$ by case 4 of $\prec_P$ again. The only question is whether $c \prec_P b$, i.e. whether $P[\theta_1, n_2, \vec{\beta}] \prec_P P[\theta_2, n_2, \vec{\beta}]$, where $\theta_1 < \theta_2$. If the deconstructors are different, then by definition there is an undecidable equivalence, since $\theta_1 <: \theta_2$; but we have assumed there are none. Therefore deconstructor(c) = deconstructor(b). And $\vec{\beta} \prec_P \vec{\beta}$ by reflexivity. So, all the conditions for case 6 of $\prec_P$ are applicable, and $c \prec_P b$.
5. This case is always $\bot$, so it doesn't apply.
6. $a = P[\theta_1, n, \vec{\alpha}], b = P[\theta_2, n_2, \vec{\beta}]$. From the rules for $\sqcap$, we know that a and b are associated with the same deconstructor and that either $\theta_1 <: \theta_2$ or $\theta_2 <: \theta_1$. If deconstructor$(c) \neq$ deconstructor(a), then there is an undecidable equivalence between c and a, since $\theta_1 <: \theta_2$ or $\theta_2 <: \theta_1$; but we have assumed undecidable equivalences don't occur. Therefore deconstructor(c) = deconstructor(a). $\vec{\alpha} \sqcap \vec{\beta} \prec_P \vec{\alpha}$ is true by induction. Therefore, $c \prec_P a$, because all the conditions of case 6 of $\prec_P$ are met.

   WLOG, $c \prec_P b$ for the same reasons.

∎

Therefore, in an OOMatch program, there can be no ambiguities at run-time other than those caused by undecidable equivalences, a class inheriting from multiple classes that are part of a set of multi-methods, or deconstructors behaving non-deterministically. Furthermore, one could write

program analyses to find even these errors in many common cases statically.

6. Implementation

OOMatch is being implemented as an extension to the Polyglot extensible compiler framework [15]. Polyglot is a compiler that translates Java to Java; it is intended to be extended so that it translates a language similar to Java to Java (which can of course be compiled to bytecode using javac).

7. Conclusion and Future Work

We have presented OOMatch, a mostly backwards compatible extension to Java that allows dispatch to take place by pattern matching. Unlike most pattern matching done in "case" statements, our matching allows the ordering of cases to be determined automatically by the compiler. It also allows regular objects to be matched, without exposing the implementation details of classes.

Some possible future work that could be done for this feature is as follows.

- There is currently no way for programmers to resolve ambiguities between methods, i.e. to specify that one method should be preferred over another. Syntax to allow manual specification of override relationships would help them fix these situations. In particular, allowing methods in superclasses to override those in subclasses (when specified) would be quite worthwhile.
- The regular parameters to a method can be deconstructed, but the implicit "this" parameter currently cannot. Special syntax to allow "this" to be deconstructed and matched would come in handy in many situations.
- A general "where" clause for a method, to allow methods to apply on any arbitrary boolean value (basically a precondition), would be useful, and trivial to add. Of course, arbitrary "where" clauses could not override each other in general, as their value is undecidable, in general, at compile time.
- Unification would be useful; i.e. the ability to re-use a matched variable later in the pattern. That is:

```
void f(Point x, x);
```

  would only be called if two equal `Point` variables are passed to `f`.
- The ability to match against variables that are in scope (e.g. instance variables) would be useful. For example, it would greatly simplify writing "equals" methods:

```
class Point {
    public Point(private int x,
                 private int y) {}
    public boolean equals(Point(x, y))
        { return true; }
}
```

- The automatic analyses to catch the possible run-time errors mentioned in Section 4.7 have yet to be implemented.
- Some form of list matching like that found in TOM [13], or even more general pattern matching (e.g. regular expressions), would further increase the power of OOMatch.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Simon Peyton Jones (editor). *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[6] Burak Emir, Martin Odersky, and John Williams. Matching Objects With Patterns. Technical Report LAMP-REPORT-2006-006, EPFL, 1015 Lausanne, 2006.

[7] 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.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 2nd edition*. Addison-Wesley, 1996. Downloaded from http://java.sun.com/docs/books/jls/.

[9] 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.

[10] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: object-oriented pattern matching for evolvable distributed systems. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 205–223, New York, NY, USA, 2003. ACM Press.

[11] 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.

[12] Robin Milner, Mads Tofte, Robert Harper, and David

MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[13] 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.

[14] 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.

[15] Polyglot Extensible Compiler Framework. Available at http://www.cs.cornell.edu/projects/polyglot/.

[16] 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.