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 an extension to Java, dubbed OOMatch. It 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.

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

OOMatch provides a new form of dispatch (determining which method to call, given a call site). It includes and subsumes multimethods (see for example [1]), but is not as powerful as general predicate dispatch [2].

The pattern matching is very similar to that found in the "case" constructs of many functional languages (ML [3], 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.

A formal semantics for OOMatch, as well as more details on the language and its implementation, can be found in our technical report [5].

2. 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.

```
abstract class Expr { ... }
class Binop extends Expr { ... }
class Plus extends Binop { ... }
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.

The key point to notice in the above example is that the second method *overrides* the first, since its pattern is *more specific than* the original's (because `Plus` extends `Expr`), and the third also overrides the first since `Binop` extends

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

`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.

3. Deconstructors

To allow the specification of patterns on objects, as in the previous section, their classes must provide a means of *deconstructing* said objects:

```
class Binop {
    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. 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.

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 hasn't been opened yet.

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))
{ ... }
```

4. Error Handling

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 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 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. These situations are discussed in [5], and a proof that there are no other situations that can cause a run-time ambiguity error is given there as well.

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. This way, the regular Java method can always be called as a last resort.

5. Implementation

The implementation of OOMatch has been done in the Polyglot Extensible Compiler Framework [4]. 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.

References

[1] 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.

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

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

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

[5] Adam Richard and Ondřej Lhoták. OOMatch: Pattern Matching as Dispatch in Java. Technical Report CS-2007-05, University of Waterloo, 2007. Available at http://www.cs.uwaterloo.ca/research/tr/2007/CS-2007-05.pdf.