OOMatch: Pattern Matching as Dispatch in Java

Adam Richard

13 Jan 2008

Introduction to OOMatch

- Implemented as an extension to Java (in Polyglot)
  - Mostly backwards-compatible with Java
- Older (Non-OO) languages - static dispatch
- Java, C++, others - overloading and receiver-based polymorphic dispatch
- OOMatch provides a more powerful means of determining dispatch, using pattern matching

Multimethods

```
void draw(Shape s) {...}
void draw(Circle c) {...}
```

- Second method overrides first, since Circle is a subclass of Shape
- Method depends on dynamic type of *all* arguments, (Java only considers the receiver argument)
- Multijava:

```
void draw(Shape s) {...}
void draw(Shape@Circle c) {...}
```

Predicate Dispatch

```
double log(double x) {...}
double log(double x)
    when x <= 0 {...}  //overrides f(double)
double log(double x)
    when x == 0 {...}  //overrides both methods
```

- Can specify arbitrary predicates or preconditions to guard entry into a method.
- When the predicate of one method m implies that of another method n, m overrides n
- Difficulty: this is undecidable at compile-time

Pattern Matching

```
match pair with
      (0, second) => second
      | _ => ...
;;
```

- OOMatch allows pattern matching not just on built-in values but on Java objects.

Motivation/Design for OOMatch

- More powerful than multimethods
- Simpler, safer than full predicate dispatch
- Provide dispatch through pattern matching of objects
  - Requires defining an ordering on the patterns

Patterns in Parameters

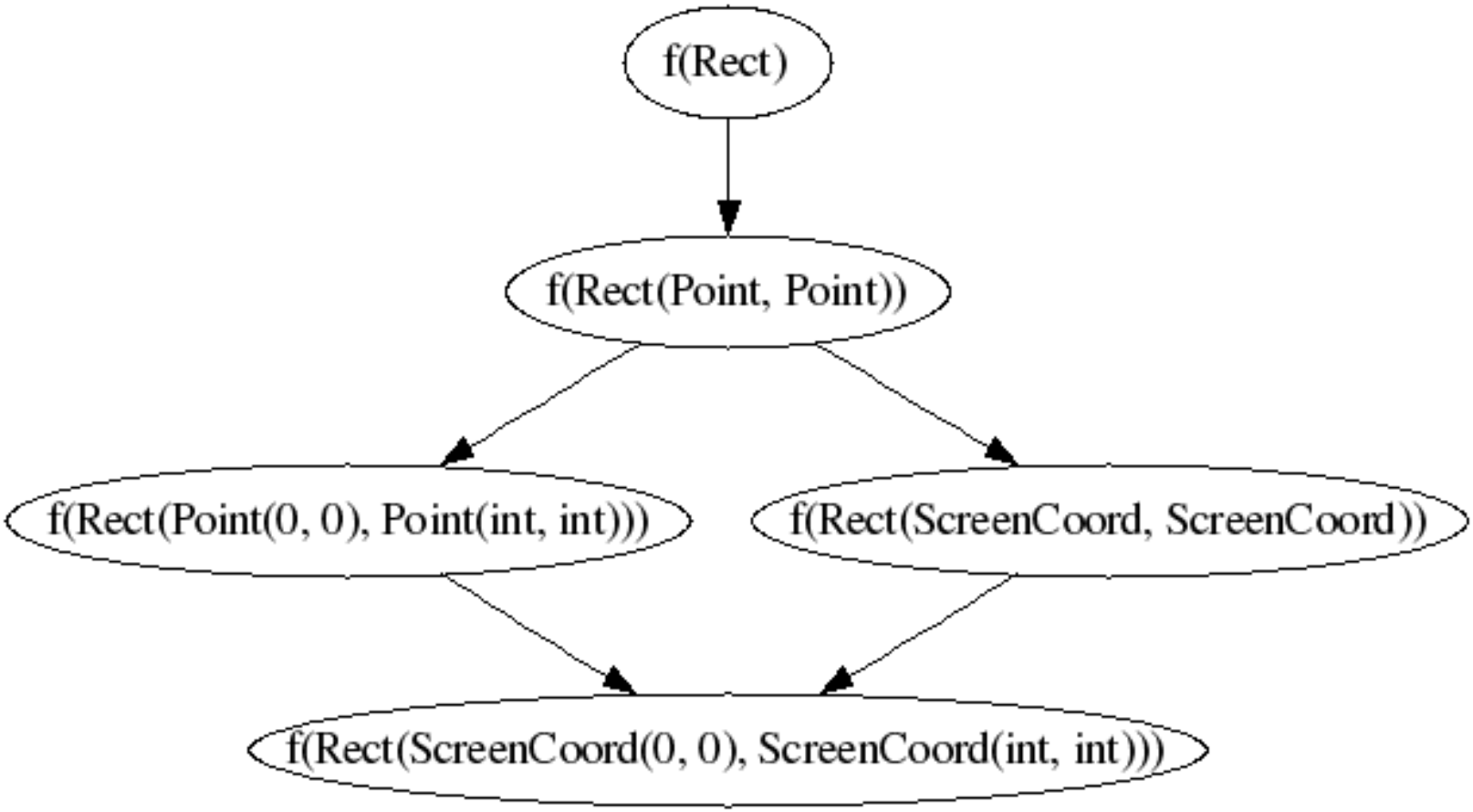
```
void f(Rect(Point(0, 0), Point p)) {}
```

- This function takes a single parameter of type Rect
- It only applies if the Rect is composed of two points, and the first point has coordinates (0, 0).
- Any named variables in the pattern (p in this case) can be used in the method body.
- Patterns can be nested to any arbitrary depth.

Overriding

```
void f(Rect r) {}
void f(Rect(Point(0, 0),
            Point(int x, int y))) {}
void f(Rect(Point p1, Point p2)) {}
void f(Rect(ScreenCoord p1, ScreenCoord p2) r) {}
void f(Rect(ScreenCoord(0, 0),
            ScreenCoord(int x, int y))) {}
```

Overriding diagram



Enabling Pattern Matching

```
public class Rect {
    private Point topLeft, bottomRight;
    ...
    deconstructor Rect(Point topLeft,
                       Point bottomRight)
    {
        topLeft = this.topLeft;
        bottomRight = this.bottomRight;
        return true;
    }
}
```

Method Overriding

- Method m_1 overrides m_2 if all of m_1's parameters are preferred over m_2's parameters
  - And they must have the same name and number of parameters
- But, methods in subclasses override those in superclasses, regardless of parameters.

```
class A {
    void f(Point p) { ... }
    void f(Point(0, 0)) { ... } }
class B extends A {
    void f(Point p) { ... } }
```

More Examples - AST processing

```
//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); }
```

More Examples - GUI event processing

```
void doEvent(Widget w) {}
void doEvent(Button("OK")) {
    //OK button clicked
}
void doEvent(Button(String name)) {
    //Process other buttons
    //(e.g. if there is an array of
    //buttons)
}
void doEvent(Textbox(String text)) {
    //Textbox event
}
```

Errors

- Sometimes multiple methods can match, but none is the most specific
- This is an *ambiguity error*.

```
//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); }
```

Run-time Ambiguities

- “Most” ambiguities caught by the compiler
- Three situations where error delayed until run-time
- This is preferable to restricting the valid OOMatch programs
- Thesis contains a proof that these three are the only situations which can cause a run-time ambiguity

Run-time Ambiguities

1. Multiple inheritance (allowed in Java for interfaces) causes difficulties

```
interface I {}
interface I1 extends I {}
interface I2 extends I {}
...
void f(I x) { ... }
void f(I1 x) { ... }
void f(I2 x) { ... }
```

Run-time Ambiguities

2. Different deconstructors in the same set of methods

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

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

Run-time Ambiguities

3. Non-deterministic deconstructors

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

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);
}
```

Use Case

- Reimplemented a class of Soot - ConstraintChecker.java in src/soot/jimple/toolkits/typing/integer/ - to use patterns.
- Multimethods eliminated the dependence on Visitors.
- Lines of Code: 1221 => 948
- Uses of instanceof: 121 => 35

Use Case - Example Simplified Code

```
//Java
if(l instanceof Local) {
    if(((Local) l).getType()
        instanceof IntegerType) {
        left = ClassHierarchy.v().typeNode(
            ((Local) l).getType());
    }
}
//OOMatch
public TypeNode left(d.Local(IntType t)) {
    return ClassHierarchy.v().typeNode(t);
}
```

Conclusion

- Pattern Matching as Dispatch is a natural next step after polymorphic dispatch and multimethods
- OOMatch - first known prototype implementation, extension of Java
- Balance between safety and flexibility explored (finding ambiguity errors vs. allowing more programs)
- Pattern Matching as Dispatch would be more useful if added to languages designed from scratch

- Website(compiler, thesis):
  `http://plg.uwaterloo.ca/~a5richar/oomatch.html`

Questions?

Let statement

```
let Point(int x, int y) = p;
```

- Matches `p` against the pattern
- An alternative to a series of calls to “getter” methods
- `x` and `y` can now be used in the method
- Match must succeed; if it fails, throws an exception

Where clause

```
void sqrt(int x) where x >= 0 { ... }
```

- Simple version of predicate dispatch - no overriding among different “where” clauses
- Method with “where” clause overrides the same method without

Non-linear pattern matching

```
void f(Point p, p) { ... }
```

- Syntactic sugar for:

```
void f(Point p, Point q)
    where p.equals(q) { ... }
```

Matching against class variables in scope

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

- Uses the same syntactic sugar for non-linear pattern matching

Manual Overriding

```
void f(Plus(int x, 0)) { ... }
| void f(Plus(0, int x)) { ... }
```

- Causes methods listed first to take precedence (as in ML, case statement, etc.)
- Rules:
  1. The methods must be able to simultaneously apply sometimes
  2. If m normally overrides n, can't write `n | m`
  3. The `|` operator causes transitive overriding

Final Methods and Cross-class Overriding

- Final means “cannot be overridden” - forces particular behavior for certain arguments

```
class BankTransaction {
    void withdraw(double amt) {
        //default implementation
    }
    final void withdraw(double amt)
        where (balance() - amt < 0)
    {
        //throw an error
    }
    ...
}
```

Static Deconstructors

```
class C {
    deconstructor Date(int year, int mon, int day)
        on java.util.Date d
    {
        year = d.getYear();
        mon = d.getMon();
        day = d.getDay();
        return true;
    }
}
class D {
    void f(C.Date(int year, int mon, int day)) {}
}
```

Motivation

- Combining functional and OO styles
  - Functional: safe, easy to work with functions
  - OO: powerful system for modularization, extending data types
- OOMatch looks at a small aspect of this: pattern matching as dispatch
  - Pattern Matching: powerful feature of many functional languages
  - Dynamic Dispatch: important OO feature to enable abstraction

Design Goals

- Freedom (no unnecessary restrictions on sensible programs)
- Simplicity (intuitive for programmers)
- Safety (generally less important than freedom)
- Retain Java's modular typechecking
- Pay only for what you use

Implementation

- OOMatch has been implemented as a Polyglot extension
  - Polyglot is a Java-to-Java compiler
  - Languages extend Polyglot to compile their language to Java
- Implementation renames all methods, inserts *dispatchers* into classes
- Deconstructors become regular methods

Backwards Compatibility

- Syntax is backwards-compatible
- Some cases where a Java program gives an error
- All differences can be caught by the compiler as either warnings or errors
- Rather than converting Java to OOMatch, OOMatch classes can use Java classes
  - Can't override Java classes with multimethods or patterns, though
- Tested compiling JEdit with OOMatch - revealed 2 errors and 19 warnings out of 50-100K LOC

Enabling Pattern Matching (Easy Way)

```
public class Rect {
    public Rect(private Point topLeft,
                private Point bottomRight){}
}
...
void f(Rect(Point p1, Point p2)) {}
```

- Combined constructor and deconstructor
- Enables matching as in the function `f`

When does a match occur?

Parameter	E.g.	Rule
Regular	Method: void f(Point p) Call: f(q)	Multimethods
Literal	Method: void f(0) Call: f(x * y)	Equality (== or .equals)
Pattern	Method: void f(ColourPoint(0, 0)) Call: f(q)	Call deconstructor on argument; match resulting list against pattern recursively.

Overriding Rules

- Formals preferred over other formals of supertype (multimethods)
  - E.g. `Circle c` $\prec$ `Shape s`
- Patterns preferred over formals of supertype
  - E.g. `Point(int x, 0)` $\prec$ `Point p`
- Literal preferred over formals which includes the value
  - E.g. `1` $\prec$ `int x`
- Patterns preferred over other patterns of supertype, same deconstructor, and recursively preferred subpatterns
  - E.g. `CPoint(0, int x)` $\prec$ `Point(int x, int y)`
- $p \prec p$

Scala

- Scala has *case classes*, which allow a class hierarchy to be fixed into a set of cases for the subclass.

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

Scala

- Case classes allow for easy pattern matching:

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

- Scala also has *extractors*, like our deconstructors
- Doesn't do pattern matching as dispatch

JPred

- Restricted, but decidable, predicate dispatch using a "when" clause.

```
void f(int x) when x > 1 { ... }
void f(int x) when x > 2 { ... }
```

- JPred computes that the second method overrides the first, because x > 2 implies that x > 1.
- Allows only built-in operators in when clause.

Visitors

```
interface Visitor {
    void caseExpr(Expr e);
    void caseStmt(Stmt e);
    ...
}
class CodeGeneration implements Visitor
{ ... }
class Expr extends Node {
    Node accept(Visitor v) {
        v.caseExpr(this);
    }
}
```