OOMatch: Pattern Matching as Dispatch in Java

by

Adam Richard

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

Abstract

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

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

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

We here present both an informal "tutorial" description of OOMatch, as well as a formal specification of the language, and a proof that the conditions specified guarantee run-time safety.

Acknowledgments

Special thanks go to my supervisor, Ondřej Lhoták, for all his help in the production of this thesis, as well as to Gordon Cormack, Peter Buhr, Brad Lushman, Nomair Naeem, and Ghulam Lashari for their help and suggestions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Object-oriented programming languages have become a widespread means of 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 is intended for such modularization.

Functional programming languages, on the other hand, have gained a group of devoted followers not only for their beauty, but also because programs written in a functional style tend to have fewer bugs. Because of their declarative nature and lack of side effects, it is generally easier to avoid bugs in a functional language (once the software passes the compiler) than in an object-oriented language. As there is a greater trend toward security, 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 checking of languages like Standard ML [MTHM97] and Haskell [PJ03] provides this safety quite well. But most functional languages (with notable exceptions such as OCaml [CMP07] and Scala [OAC$^+$06]) 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. 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 composable with each other, so that different pieces of code written in different styles can be pieced together to form a program.

One might counter that we could simply write separate components in separate languages and combine the components with a standard interface. A functional or object-oriented language could be selected for each component depending on which is most applicable to the programmer or situation. The problem with this approach is that we might still want to use elements of both styles at once. We might want a class, with all the power of an object-oriented class, that contains a function that can be used as a value; or we might want a function that contains a nested function as well as a class definition within it. In general, the combination of several styles may yield new and powerful idioms or code patterns that are not present in their mere sum, and by studying the combination of functional and object-oriented styles, we hope to discover what some of these idioms might be.

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

Further, and most importantly, our pattern matching is used in determining method dispatch. The patterns are specified as parameters to methods (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. Methods with more specific parameters override methods with more general parameters. Since parameters of subclass type are considered more specific than those of superclass type, this feature subsumes polymorphic dispatch and multimethods. 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 simply use if-else blocks and casting instead of pattern matching. The practical value of pattern matching as dispatch would then be lost.

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

The current implementation of OOMatch has been done in the Polyglot Extensible Compiler Framework [POL]. Polyglot translates to Java, but contains all the functionality of compiling the base Java language, which prevents implementers from having to write a compiler from scratch. It is therefore useful for writing prototype compilers for new Java-like languages.

Some of the design goals of OOMatch are as follows.

- Flexibility. The language should not unnecessarily prevent programmers from writing code that may make sense. When a choice must be made between preventing some programs that make sense and allowing some erroneous programs, we normally choose the latter.
- Simplicity. Our dispatch feature involves a lot going on in the background to determine when a method should be called; the programmer should rarely need to pay attention to this. The syntax and rules should be such that they can see intuitively what the program does.
- Safety. Our feature introduces new potential errors, and the compiler should report on as many of these as possible.
- Modular typechecking. Java has the ability to compile a class without knowledge of all the classes that might use or extend it. OOMatch should have this ability as well.
- Programmers should only have to pay for the features they use [Str97]. If they use the OOMatch compiler, but do not use our pattern matching or overriding features, it should not hinder performance.

Of course, many of these goals are at odds with each other. Throughout this thesis, we explain why we chose one option when a choice involving a tradeoff is necessary.

Though OOMatch is implemented as an extension to Java, it would likely be straightforward to adapt the feature to other object-oriented languages with receiver-based dispatch, such as C++ [Str97].

The rest of the thesis is organized as follows. Chapter 2 gives background and related work leading up to OOMatch and should be helpful to those unfamiliar with all the concepts used. Chapter 3 gives an informal description of the language and its various features. Chapter 4 gives a formal specification of the core of OOMatch (the pattern matching dispatch). It also describes the checks done by the compiler, and proves that if an OOMatch program compiles, there are only certain given conditions under which a run-time error can occur. Chapter 5 describes the current prototype implementation of OOMatch, and gives the main algorithms used to implement the dispatch present in it. Chapter 6 describes some of the main limitations of the compiler as it currently exists. Chapter 7 describes some ways in which OOMatch is useful in real situations. Chapter 8 discusses other related work that is best discussed after the reader understands OOMatch. Chapter 9 concludes and lists future work that could be done on OOMatch.

Chapter 2

Background

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

2.1 Pattern Matching

Pattern matching is a popular feature in functional languages. Suppose one is using a language without pattern matching, like Java. Suppose they need to test whether a pair of integers is 0, and if so, return the second element of the pair. The Java code to do so could look like this:

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

In other words, accessor functions and comparisons are necessary to get the components of a structure and test them for the relevant conditions. Note, too, the separation of the test for 0 and the extraction of the second element.

This type of task could be simplified in a language with pattern matching. For example, in SML [MTHM97], it might look something like this:

```
case pair of
```

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

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

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

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

2.2 Dispatch

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

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

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

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

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

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

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

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

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

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

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

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

2.2.1 Multimethods

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

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

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

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

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

But this is incorrect because the version of "equals" shown does not in fact override Object's "equals" method, which has signature:

```
public boolean equals(Object obj)
```

[JAV]. Because the `equals` in `C` does not have the same parameter types, the methods become overloaded rather than overridden. This means that, for example, this code:

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

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

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

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

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

2.2.2 Predicate Dispatch

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

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

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

Chapter 3

Using OOMatch - Informal Description

3.1 Pattern Matching

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

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

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

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

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

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

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

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

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

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

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

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

The `0` in the second method means that the pattern is only matched when the numeric constant's value is 0. The named variables in the patterns are given the value that is matched, so that this value can be used by referring to the declared name in the method body. 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 an arbitrary depth. The most specific match is always chosen first. So, for example, we could add another method with signature

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

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

An interesting exercise is to think of what would be necessary to rewrite `optimize` in pure Java. The programmer would basically have two options: put every case in one method, or have separate methods, with different names, and a method to select which of these to execute. Here is what the second choice might look like:

```
Expr optimize(Expr e) {
    if (e instanceof Binop) {
        Binop eAsBinop = (Binop)e;
        Expr op1 = eAsBinop.left();
        Expr op2 = eAsBinop.right();
        if (e instanceof Plus) {
            Plus eAsPlus = (Plus)e;
            if (op2 instanceof NumConst) {
                NumConst op2AsNumConst = (NumConst)op2;
                if (op2AsNumConst.value() == 0)
                    return optimizePlusZero(op1);
            }
        }
        if (op1 instanceof NumConst &&
            op2 instanceof NumConst)
        {
            return optimizeFold(eAsBinop,
                (NumConst)op1, (NumConst)op2);
        }
    }
    return e;
}

Expr optimizePlusZero(Expr e) { return e; }

Expr optimizeFold(Binop op, NumConst c1, NumConst c2) {
    return op.eval(c1, c2);
}
```

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

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

Alternatively, since it does not matter which method is called in this case, the user can specify manually that the "+ 0" optimizations overrides the constant folding one simply by inserting the `|` operator between them. (This feature is described in Section 3.11.)

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

For example, consider the following method:

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

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

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

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

3.2 Deconstructors

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

In OOMatch, access specifiers can be added to constructor parameters:

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

The `public` specifier on parameters does 4 things:

- It declares the variable to be a public instance variable of the class. If there already is an instance variable of that name, a duplicate definition error occurs.
- It declares a parameter to the constructor.
- It assigns the argument passed to the constructor to the instance variable. This assignment happens at the *end* of the constructor body.
- It allows the object to be deconstructed in a pattern that corresponds to the way it was constructed.

If there are multiple constructors with the same instance variables declared as parameters, no error occurs; they each construct the same variable.

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;
        return true;
    }
    ...
}
```

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

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

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

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

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

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

The second method above overrides the first.

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

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

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

3.3 Order of Deconstructors

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

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

3.4 Null

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

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

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

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

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

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

3.5 Undecidable Errors

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

3.5.1 Interfaces

The first potential cause of ambiguity is caused by multiple inheritance, which is partially allowed in Java by implementing multiple interfaces, or by extending a class and implementing an interface. Consider the following trivial pair of methods:

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

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

A simple test showed that JPred [Mil04] 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 was migrated to a language with multiple inheritance, the problem would return.

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

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

This causes `f(A)` to be chosen and `f(B)` to be removed from consideration.

3.5.2 Different deconstructors

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

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

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

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

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

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

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 two deconstructors might be returning completely different values, 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.

3.5.3 Non-deterministic deconstructors

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

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

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

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

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

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

3.6 Where clause

OOMatch provides a general “where” clause to enable arbitrary applicability conditions. For example, the following pair of functions compute the absolute value of a number:

```
double abs(double x) where (x < 0) { return -x; }
double abs(double x) { return x; }
```

The first method only applies when its boolean condition evaluates to true. Any arbitrary boolean expression is allowed in a where clause, including method calls. The “where” clause in OOMatch is much simpler than in most languages with predicate dispatch. In particular, there are no override relationships among different “where” clauses; this would require evaluating whether one clause implies another, which is undecidable in general. Extensive outside work on doing this evaluation for a subset of the possible boolean expressions has already been done (by [Mil04], for example). Instead, in OOMatch, a method with a “where” clause always overrides the same (or a more general) method without the “where” clause. Hence, the first `abs` method above overrides the second.

In combination with the `inc` keyword (see Section 3.1), a where clause can also be used to add arbitrary preconditions to a method. For example, the following method computes a square root, and gives an error if passed a negative number:

```
inc double sqrt(double x) where (x >= 0)
{ ... }
```

Note that the parentheses around the “where” expression are necessary; without them, parsing the “where” clause would be more difficult. In particular, “where” clauses that contain anonymous classes would result in an ambiguity. Suppose the parser sees the following code:

```
void f() where new Object() {
```

At this point, the parser does not know whether the `{` is the start of a method, or the start of an anonymous class declaration extending `Object`. It could be argued that the former is never the case in useful code, since a new expression is not a boolean expression, but getting the parser to understand that is potentially difficult. Hence, to keep our implementation relatively simple, we decided to require brackets around the “where” expression for now.

Translation of the “where” clause simply involves creating a separate method, with boolean return type, which evaluates and returns the expression. The method only applies if this method returns true.

Note that, as with deconstructors (see Section 3.3), the order in which methods are checked for applicability is left unspecified. It is up to the programmer to ensure that the evaluation of one “where” clause does not affect the result of another.

3.7 Matching Against the Values of Variables

OOMatch allows non-linear pattern matching – that is, a variable defined in a pattern or in the parameter list can appear again in the parameters. For example:

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

The reference to p in the second parameter means that f is a method accepting two Points, but it only applies if the two arguments are equal. This is a fairly simple feature, and in fact is merely syntactic sugar for a "where" clause that checks for equality. Equality is determined by the == operator for primitive types, and the equals method for objects. The above code is desugared to something like the following (eliding null checks):

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

where q is a fresh name that is guaranteed to not clash with any other name.

In fact, OOMatch allows variable references in patterns that reference external variables, as well - namely, the instance variables of the class. Perhaps the most useful motivation for allowing the matching of instance variables is that it could allow for very concise equals methods (assuming the Java standard library is recompiled as OOMatch code). Here is an example:

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

The pattern Point(x, y) matches only Points whose components (returned from the deconstructor) are equal to the instance variables x and y, respectively, of the "this" object. Compare the equivalent Java code for Point.equals:

```
public boolean equals(Object other)
{
    if (!(other instanceof Point)) return false;
    Point otherPoint = (Point)other;
    return otherPoint.getX() == x &&
        otherPoint.getY() == y;
}
```

Section 6.4 shows that this way of writing equals does not work unless the Java standard library is migrated to OOMatch. This example is shown only as an illustration of how matching against the values of variables could be useful.

3.8 Cross-class Ambiguities and Final Methods

An issue that arises when studying multimethods is whether the receiver should take precedence over the other parameters, or whether it should be treated the same as any other parameter. The latter involves giving an ambiguity error. For example, consider this code:

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

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

The problem is that the `Circle(0, 0)` pattern is more specific than `Shape`, but there is a method in a subclass with the more general pattern `Shape s`. Suppose then that this call occurs:

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

We have two options: give an ambiguity error, or resolve the ambiguity in favor of one of the methods. In OOMatch, we decided to resolve the ambiguity in favor of `Square.intersection`, by default – i.e., to make Java overriding take precedence over OOMatch overriding. This is because the Java behaviour - subclasses overriding superclasses - is probably familiar, and most commonly the `Square.intersection(Shape)` method is probably intended to mean "this is the new way to handle all `Shape`s for this method". Giving an ambiguity error would disallow programmers from declaring that fact.

The user may sometimes want the reverse behaviour. Perhaps a class writer wants to impose a fixed method to handle a particular set of arguments; in other words, to have that method take precedence over any method the user provides in the subclass. Hence, it would be nice to provide a way to specify that a method takes precedence over subclass methods. We have chosen to provide a simple means of attaining this ability through the use of "final".

First, to understand the motivation, consider the following example. Suppose you are writing a class for a bank transaction, which may have several subclasses, each representing a different type of transaction (Internet, phone, teller), each of which requires special processing, logging, etc.. You want to impose the rule that no matter what method is used to do a withdrawal, attempting to withdraw more money than is in the user's account results in an error (a fairly important condition to be able to ensure). Using final overriding, this rule might be imposed something like this:

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

Now subclasses of `BankAccount` are free to override the withdraw method, but if their method is ever called in a situation where the "where" condition is true, the error is always given.

In Java, a "final" method is one that can not be overridden - that is, there cannot be a method in a subclass with the same parameters. In OOMatch, "final" means that there can not exist a method that would take precedence over the final method - either in the same class or by a method in a subclass. The final method takes precedence over all methods in subclasses which have more general parameters than the final method. To help understand how this works, you can treat a final method as one that is copied into all descendent classes.

To understand the motivation for this rule, suppose it were not in place - suppose a method in a subclass took precedence over a final method in a superclass. Suppose also that the method in the subclass handles a *superset* of the final method's parameters. Since this method is kind of "overriding" the final method, in that it is called in place of it in some cases, should an error be given? That would not really make sense, because then subclasses would be prevented from providing an operation that happens to include the (potentially small) set of values that the final method handles. What the programmer really intends by labelling a method m final is, “This is the only way to do m. You cannot provide a different way to do m.” Hence, in the context of OOMatch, it makes sense for final methods to take precedence over methods in a subclass.

It is interesting to note here that this feature is solving what could be considered a cross-cutting concern, normally solved using Aspect-oriented programming [KLM+97].

3.9 Abstract Methods

Abstract methods may not contain patterns. The reason is that their purpose is to provide an interface for the client, and patterns are part of the implementation of a method. Since interfaces only contain abstract methods, patterns can never appear in them.

However, *partially abstract methods*, as described in [Mil04], are potentially very useful, and are allowed in OOMatch. A partially abstract method is an abstract method that is accompanied by concrete implementations to handle certain cases of the operation. Each concrete implementation is represented as a method that overrides the abstract method. The `withdraw` method given in Section 3.8 would likely be better made abstract (and the `BankTransaction` class made an abstract class), with the error case providing a special case for it.

Note that it never makes sense for a non-final method to override an abstract method in the same class. The reason is that the abstract method, which is always in an abstract class, is forced to be implemented in any concrete subclass. But this method, by the rule that methods in subclasses override those in superclasses, always overrides the non-final method in the abstract class. And since abstract classes cannot be instantiated, the non-final method could never be called from outside the class (the only situation in which overriding is useful). Therefore, it is a compile error to override an abstract method with a concrete implementation that is not labelled “final”.

3.10 Overriding among Primitive Types

There are three ways in which overriding occurs for parameters of primitive type. We give an intuitive description of the rules here; a formal specification is given in

Chapter 4. The general idea behind these rules is that "more specific" values, or sets of values, override "less specific" ones. Consider the following example:

```
class C {
    void f(double x) { ... }
    void f(int x) { ... }  //overrides f(double)
    void f(0.0) { ... }  //overrides f(int) as well as f(double)
    void f(0) { ... }  //overrides f(0.0)
}
```

The example illustrates the three rules for primitives types and values:

- First, regular parameters of primitive type override larger ones. Hence, "int" overrides "double" in the example.
- Second, literal values can override regularly declared formal parameters of primitive type, if the value is within the set of values that the type represents. Hence, `0.0` is considered more specific than `int`. Note that the value need not be of the type of the parameter to override it. As long as there exists a value equal to `0.0` that is of type `int`, overriding occurs.
- Finally, a value X can override another value Y, if X and Y are equal but Y is of a more general type. Hence, `f(0)` overrides `f(0.0)` above. This is not useful in this case, because `f(0.0)` can never be called, but it is permitted for completeness' sake.

3.11 Manual Overriding

Though the automatic overriding determined by the compiler is often useful, sometimes a user might want to manually specify an ordering on the methods. This is especially useful to resolve an ambiguity without writing extra code. Simply putting the Java `|` operator (used as the bitwise OR operator in Java) between two methods causes the first one to override the second. For example:

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

does not cause the usual ambiguity error; the first version of `f` is called whenever both versions apply. Several methods can also be strung together in a sequence separated by `|`.

To avoid cycles in the override relationships and other difficulties, there are a few rules governing the use of this feature:

- It must be possible for methods separated by `|` to simultaneously apply for some arguments. (What it means for methods to possibly apply simultaneously is formalized later in Section 4.5.2).
- If the declaration m_1 `|` m_2 appears, then it is an error if m_2 would normally override m_1.
- If the declaration m_1 `|` m_2 appears, then any method m that overrides m_1 now overrides m_2 as well.

More powerful forms of manual overriding were considered as well. In particular, another possibility would be to allow the programmer to declare precedence orderings on the methods, similar to the `%prec` feature in Yacc [Joh79]. The `|` operator was decided on because it is simple, easily understandable, and familiar to programmers with an ML background. If it is discovered that a more powerful way to specify overriding would be sufficiently useful, one could be added at that time.

3.12 Let statement

Sometimes users want to decompose an object into components without doing dispatch. For this, OOMatch provides a simple "let" statement for pattern assignment. It is inspired from the let statement found in ML, which allows pattern matching to be performed in an assignment. An example of its usage is:

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

This extracts the components `x` and `y` from `p`, without the need to call any accessor methods. If designer of the `Point` class desires, they could even omit the accessor methods from the class and force the use of pattern matching to extract the components of the point. The left hand side of the assignment must be a pattern, and the right hand side an expression of the type of the pattern. If the match fails, an exception is thrown (of type `java.lang.Error`).

3.13 Static deconstructors

There are many Java classes already in existence, particularly in libraries (which the programmer can not modify), on which it may be convenient to do pattern matching. But pattern matching requires a deconstructor, and it may be impossible or undesirable to put a deconstructor in those classes. To allow such classes to be matched in patterns, OOMatch provides a feature called *static deconstructors*.

A static deconstructor is simply a deconstructor that decomposes an object of a type other than the one it is declared in. A special "on" clause is used to declare the object being deconstructed. Here is an example:

```
class C {
    deconstructor Rect(Point topLeft, Point bottomRight)
        on Rect r
    {
        topLeft = r.getTopLeft();
        bottomRight = r.getBottomRight();
    }
}

class D {
    void f(C.Rect(Point p, Point q)) {
    }
}
```

`Rect` is treated like a static method and, in fact, is translated as such. The type of the parameter to `f` is `Rect`, not `C`. If a method containing a pattern that uses the `Rect` deconstructor were in the class `C`, it would not need to qualify the deconstructor with "`C.`".

Also note the need to call accessor methods of the `Rect` class, which would not be needed for a deconstructor directly in the `Rect` class. This is necessary because `C` does not (and should not) have access to the private data of `Rect`. This means, of course, that static deconstructors require the objects they are deconstructing to provide methods that extract their relevant components.

3.14 Backwards Compatibility

OOMatch is mostly backwards compatible with Java. The syntax of Java is a subset of OOMatch; that is, all Java code passes the OOMatch parser. However, there are four cases where a valid Java program gives a semantic error when treated as an OOMatch program, and one case where code has a different semantics in OOMatch than in Java.

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 example 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 the following invocation could cause a run-time ambiguity:

```
C c = new D();
c.f(new B(), new B());
```

Second, the meaning of a Java program might be slightly different when treated as an OOMatch program. In particular, methods that 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` invokes the first version of `f`, despite the fact that the argument's dynamic 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 behaviour may seem stranger, there may be legacy code that depends on it, and whose behaviour should not be changed. Nevertheless, this disadvantage was deemed a worthwhile price to pay to avoid the need for specific syntax for multimethod behaviour (such as the "@" symbol from MultiJava [CLCM00]). Special syntax would make the feature more cumbersome and confusing to learn, which, we felt, is worse than violating backwards-compatibility.

Third, OOMatch requires methods that could possibly apply to the same call site to have the same return value and throws clause, although in many cases only a subset of the throws clause needs to be in the overriding method. (This is specified in more detail in Section 4.5.2.) Java is less restrictive here, making no such requirement for overloaded methods. There may be overloaded methods with completely different return types or throws clauses that become overridden as OOMatch methods, and this would cause a compile error. For example:

```
//OK in Java; error in OOMatch
class C {
    int f(Shape s) { ... }
    String f(Circle s) { ... }
}
```

Finally, the restrictions on overriding "final" methods from Sections 3.8 and 3.9 are not present in Java. Hence, the following valid Java code fails to compile with OOMatch:

```
final void f(int x) {}
void f(short x) {}  //can't override a final method
```

as does this:

```
abstract void f(int x);
void f(short x);  //must be final
```

It should be noted, despite these differences, 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. OOMatch files have the extension ".oom", and so can be easily distinguished from legacy Java code. When assurance is needed that legacy code retain its exact behaviour, or when Java code that does not compile with OOMatch cannot be changed, those parts of the program can be compiled with the Java compiler. Note, though, that Java code should not import or use OOMatch code.

Also, the current implementation contains an option to give warnings for cases where the semantics between the Java and OOMatch version of a program would differ. Hence, the compiler can detect and inform the programmer of all potential cases of backwards-compatibility violations.

To get some idea of how serious the backwards-compatibility difficulties are, we attempted to compile an open-source Java program, JEdit 4.2 [JED], with the OOMatch compiler. JEdit is a programmer's editor consisting of 394 Java files containing about 50-100K lines of code. The compiler gave no errors regarding improper use of `final` and no ambiguity errors.

The compiler gave 2 errors related to return types of methods that might simultaneously apply and no errors for different "throws" clauses. There were also 19 warnings related to possibly different semantics due to multimethods. Many of these warnings were due to methods with different primitive types as parameters, such as:

```
void set(int) { ... }
void set(long) { ... }
```

These methods are not a problem, because OOMatch uses Java's method invocation semantics (described later in Section 4.4.1), so the static type of the argument must still be a subtype of the primitive parameter type in order for the method to be called. Also, many of the cases of possibly different semantics will not necessarily change the behaviour of the program. The sparsity of the warnings is an indication that it is probably relatively rare (in a language without multimethods) to overload methods where one method handles a subset of the parameters of the other.

There were also many errors related to the JEdit code (now considered OOMatch code) using Java code (namely, the Java standard library), which are discussed later in Section 6.4. These errors would not be a problem if the standard library were also recompiled to OOMatch.

3.15 Multi-threaded Applications

Using patterns in concurrent programs may introduce extra issues that the concurrent programmer should be aware of. The difficulty is that, in the process of dispatching to a method, deconstructors may be called and may read various components of the object. (It is possible for them to write these components, as well, but good practice says that a deconstructor should not do so.) Therefore, if the argument to the method is shared among multiple threads, it could happen that the argument, or one of its components, is being modified by another thread while the current thread is in the process of dispatch. This situation causes incorrect dispatch semantics, and possibly a crash. It is up to the programmer to prevent this, by using locks or some other means to prevent a race condition on calls that (may) use deconstructors, and writes to the components in those deconstructors. Alternatively, the safest approach would be to never use pattern matching on shared variables.

To illustrate this problem, suppose there is the following class:

```
class C {
    void f(X x) { ... }
    ...
}
```

Suppose there is also a call to `f`, for a variable `a` that is shared by multiple threads:

```
f(a);
```

Finally, suppose there is a statement such as the following, which another thread could execute simultaneously with the above call:

```
a = new X();
```

A Java programmer is accustomed to the need to make the call and assignment mutually exclusive, so that they cannot be executing simultaneously. However, suppose the class X has several subcomponents, and a statement such as the following could execute simultaneously with the call to `f`:

```
a.b.c = new Z();
```

In Java, there is not necessarily any danger of a race condition here, because Java is pass-by-reference. Only the *reference* to `a` is being read during the call to `f`. The assignment of `a.b.c` is only *reading* `a`, and simultaneous reads of a shared variable are no problem. But suppose an OOMatch programmer now adds the following new method to either `C` or, worse, a subclass of `C`:

```
void f(X(Y(Z(int i)))) { ... }
```

Assume `Y` is the component corresponding to `a.b`, and `Z` is the component corresponding to `a.b.c`. Then the possibility of a race condition has now been introduced. There may be a read of `a.b.c` in the dispatch code of the new `f`, happening simultaneously with a write to `a.b.c` in its assignment statement. Therefore, calls to `f` need to be made mutually exclusive with the assignment to `a.b.c`, in addition to assignments to `a`, where they did not before. This problem is a particularly difficult one because the new method containing patterns may have been added without the knowledge either of the class `C` or the call to `f`.

The problem would be easier to avoid if there were a way to make dispatch atomic, or atomic over modification to a group of objects. But there is no way to do this in Java. Putting the dispatch code in a synchronized block does not solve the problem, because `synchronized` only prevents multiple threads from being in the dispatch code at once - there is still nothing preventing an outside thread from modifying the same objects. There are new languages with this feature, such as AtomoΣ[CMC+06], but they are not as stable or readily available (as atomic blocks are a fairly new research area). Incorporating atomic blocks, or a translation to a language that uses atomic blocks, into the OOMatch compiler has therefore been left as future work. It should be noted that the concurrency issues introduced by OOMatch can be dealt with by the programmer, generally without much difficulty, and that they are no more difficult to solve than many other concurrency issues that atomic blocks could also solve. The important thing for the OOMatch programmer to understand is that there are extra situations where they need locks where they previously did not, and if they choose to pass shared variables to patterns, they should understand what these issues are.

Chapter 4

Formal Specification

We now specify precisely how the core of OOMatch works. First the core syntax is given in Section 4.1. Then we define some mathematical objects in Section 4.2, which are used to conveniently and concisely refer to the important OOMatch constructs. In Sections 4.3 and 4.4 we describe the semantics of OOMatch, namely how deconstructors are interpreted and which method is called for a given call site. Section 4.5 describes the important checks the compiler does to determine whether the input is a valid OOMatch program. Finally, Section 4.6 specifies the conditions under which a run-time ambiguity error cannot occur, given that the safety checks in Section 4.5 succeed, and proves that the conditions specified indeed prevent run-time ambiguity errors.

4.1 Grammar

The grammar for the syntax of the key features of OOMatch is as follows, eliding unimportant details such as access specifiers and syntactic sugar. It is an extension of the Java Language Specification, second edition [GJSB96]. New nonterminals are given in bold. These symbols are found in the body of the JLS, chapters 3 and 8.

$$
\begin{aligned}
\textit{ClassMemberDeclaration} ::= \;& \textit{FieldDeclaration} \\
|\;& \textit{MethodDeclaration} \\
|\;& \textit{ClassDeclaration} \\
|\;& \textit{InterfaceDeclaration} \\
|\;& \boldsymbol{\textit{Deconstructor}}
\end{aligned}
$$

Deconstructor ::= `deconstructor` *Identifier*
`(` *FormalParameterList*$_{opt}$ `)` *MethodBody*
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)`.

Note that patterns may only appear in methods, not constructors. This is discussed in Section 6.1.

4.2 Notation and Definitions

The following notation is used to refer to OOMatch elements:

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

Let $T <: T'$ be used to state that the Java type T is a subtype of the Java type T'. Since Java 1.4 does not define subtyping between primitive types, we need to define this precisely. The primitive type subtype relations are the same as in Java 1.5.

Definition 4.2.1 *1. $T <: T'$ if T and T' are classes or interfaces and T extends or implements T'.*

2. null $<:$ T if T is a class, interface, or array type.

3. `byte` $<:$ `short` $<:$ `int` $<:$ `long` $<:$ `float` $<:$ `double`*, and* `char` $<:$ `int`*.*

4. For array types, `A[]` $<:$ `B[]` *if $A <: B$.*

5. $T <:$ `Object` *for all class, interface, and array types T.*

6. $T[] <:$ `Cloneable` *and $T[] <:$* `java.io.Serializable` *for all array types $T[]$.*

Finally, $<:$ is transitive, i.e. $a <: b$ and $b <: c \Rightarrow a <: c$, and reflexive, i.e. $x <: x$, for all Java types a, b, c, and x.

Lemma 4.2.1 *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. [GJSB96] ∎

4.3 Deconstructor disambiguation

The deconstructor invoked for a given pattern parameter is fixed at compile time. Deconstructors may be overloaded, i.e. there may be multiple deconstructors in a class or class hierarchy with the same name but different parameters. But if this is done, then for any deconstructor reference in a given method header, there must be a unique most specific deconstructor that fits the reference. The compiler attempts to find exactly one deconstructor definition to match the reference; if it can not, there is a compile-time error. We now define formally how it determines this.

First, we need an auxiliary function *type* that gives a Java type corresponding to a parameter:

Definition 4.3.1

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

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

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

Next, we define the conditions under which a deconstructor $D[n, \vec{T}]$ is *eligible* for a pattern $P[T_r, n_2, \vec{\theta}]$. It is eligible if all these conditions hold:

- The deconstructor is in T_r or one of its supertypes
- $n = n_2$ (The name referenced by the pattern is the same as the deconstructor's name)
- $|\vec{T}| = |\vec{\theta}|$ (The pattern has the same number of subpatterns as the parameters in the deconstructor)
- $\mathit{type}(\theta_i) <: T_i$ for all parameters i (The types in the pattern are a subtype of the deconstructor's parameters)

If no deconstructors are eligible for a pattern, a compiler error occurs. If there are multiple eligible deconstructors (i.e. if there is overloading or "overriding" among the deconstructors), the compiler tries to find the most specific one. A deconstructor $d_1 = D[n, \vec{T}]$ is more specific than $d_2 = D[n, \vec{U}]$ if $T_i <: U_i$ for all parameters i, and if d_1 is in a subtype of the class d_2 is in (or if they are in the same class).

If there is a unique most specific deconstructor among all the eligible deconstructors, that one is selected for the pattern; otherwise, a compile error occurs.

4.4 Method dispatch

We now examine the question of which method is chosen to call for a given call site. First, we define when a method is *applicable* for a call site - that is, when it is a potential candidate to call. Then we define an ordering on the methods, specifying which are preferred. Finally, we specify how the compiler determines the method to call from this information.

4.4.1 Applicable methods

We now define a predicate $applicable(M[T_r, n_M, \vec{p}, \vec{T_t}], n, \vec{a})$, which takes a method with name n_M and parameters $\vec{p}$, and a method call on a method named n with argument types $\vec{a}$. The predicate returns true (i.e. the method is applicable for the call) if all of the following conditions hold:

1. $n_M = n$ (The method's name is the same as the name of the method being called.)
2. $|\vec{a}| = |\vec{p}|$ (The number of arguments matches the number of parameters in the method.)
3. Each argument is *admissible* (a predicate defined next) to its corresponding parameter: $admissible(a_i, p_i)$ for all parameters i.

The predicate $admissible(a, p)$ determines whether an argument a may be passed to a parameter p. The following cases specify when it holds:

- If p is a regular Java formal, and a is not null, then admissibility is determined according to the Java method invocation-convertible rules: If a is method invocation convertible to the type of p, then *admissible* is true. [GJSB96, Section 5.3] If p is a regular Java formal and a is null, then the *static* type of a is considered. If the static type of (the null value) a is a subtype of the type of p, then *admissible* is true.
- If p is a literal $C[v, T]$, then admissibility holds if a is equal to v. If v is a primitive value or null, equality is determined with the Java `==` operator. Otherwise, if it is a String literal (the only possible literal of Object type), equality is determined by checking whether `v.equals(a)`.

- If p is a pattern $P[T_r, n, \vec{p}]$, then admissibility is determined by checking whether the runtime type of a is a subtype of T_r; if not, a is not admissible. Next, the deconstructor associated with p (see Section 4.3) must be executed on a to determine admissibility. If the deconstructor returns false, a is not admissible to p; otherwise, the deconstructor produces a set of values $\vec{a'}$ that apparently represent the components of a. Admissibility is then true if for all i, $\mathit{admissible}(p_i, a'_i)$ is true (that is, admissibility is checked recursively).

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

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

4.4.2 Preferred Methods

We now specify how to determine override relationships among methods. A method $m_1 = M[\_, n, \vec{p}, \_]$ is preferred over $m_2 = M[\_, n, \vec{q}, \_]$, denoted $m_1 \prec_M m_2$, if either of the following conditions hold:

1. m_1 is contained in a subclass of m_2, or
2. m_1 and m_2 are in the same class, have the same number of parameters, and $p_i \prec_P q_i$ for all parameters i (All the parameters in m_1 are preferred over those in m_2). $\prec_P$ is defined next.

The predicate $\prec_P$, which determines whether one parameter is preferred over another, is defined (recursively) to hold in the following cases:

1. $F[T_1] \prec_P F[T_2]$ whenever $T_1 <: T_2$.
2. $C[v, \_] \prec_P F[T]$ whenever the Java expression `(`T`)` v `==` v evaluates to true.
3. $C[v_1, T_1] \prec_P C[v_2, T_2]$ if the Java expression v_1 `==` v_2 evaluates to true, and $T_1 <: T_2$, where T_1 and T_2 are the types of the literals v_1 and v_2.
4. $P[T_1, n, \vec{p}] \prec_P F[T_2]$ when $T_1 <: T_2$.

5. $P[T_1, n_1, \vec{p_1}] \prec_P P[T_2, n_2, \vec{p_2}]$ when $T_1 <: T_2$, both patterns are associated with the same deconstructor, and $\forall i. p_{1_i} \prec_P p_{2_i}$.

Further, $\prec_P$ is defined to be a preorder, i.e. reflexive and transitive.

Lemma 4.4.1 *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 condition 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. $a = P[T_1, n, \vec{p}], b = F[T_2]$. The condition is always false because $F[T_2] \not\prec_P P[T_1, n, \vec{p}]$ for any formal and pattern parameters, so the implication is true by default.
5. $a = F[T_1], b = P[T_2, n, \vec{p_2}]$. Without loss of generality, 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 $a = b$.

∎

4.4.3 Overall Method Dispatch

OOMatch selects a method for a given call site as follows. It first gathers the set of all methods from the class of the run-time type of the receiver object, and its superclasses, that are applicable to the call. If no such methods are applicable, a no-such-method error occurs at runtime. Otherwise, from the group of methods that applied, there should be a unique method that is preferred over all the other applicable methods. If there is not, an ambiguity error occurs at runtime. If there is, that method is called.

Note that, because $\prec_M$ is antisymmetric, it is impossible for multiple methods to be preferred over all methods, i.e. there can be no cycles in the override relationships.

In Section 4.6.4, we give a set of conditions that, if true, guarantee that ambiguity errors and no-such-method errors cannot occur at run-time.

4.5 Compile-time checks

We now describe the checks an OOMatch compiler is required to make to prevent many cases of the errors mentioned in Section 3.5.

4.5.1 Parameter Intersection

We would like it to be the case that whenever more than one method might apply to a call, there exists a preferred method that handles the intersecting cases. To help ensure this, we first need to define an operator representing the intersection of parameters. The intersection operator $\sqcap$ takes a pair of parameters, and returns a parameter representing their intersection, or is undefined if it can not determine that the parameters intersect. We would like to define $\sqcap$ such that these two conditions hold:

1. If two methods can be simultaneously applicable, then the intersection of their parameters should be defined. If they can not be simultaneously applicable, then their intersection should be undefined.
2. The intersection of two parameters should be preferred over both parameters, i.e. $p_1 \sqcap p_2 = p_3 \Rightarrow p_3 \prec_P p_1$ and $p_3 \prec_P p_2$.

Thus, loosely, if the intersection of every pair of methods (when defined) exists as a method in the program, ambiguity errors are prevented. There are a few exceptions, which we formalize later in Section 4.6.4.

The specific intersection function that we claim satisfies the above properties is defined next.

Definition 4.5.1 *Several cases of the parameter intersection function $\sqcap$ are shown in Table 4.1. The function is defined to be symmetric; thus, the blank entries in the table correspond to entries opposite the diagonal.*

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

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

4.5.2 Conditions to be checked statically

We now define the conditions under which a class with its set of methods is considered valid or well-formed. Consider a class C, which is valid by the Java rules, and let M_C be the set of methods in C. All of the following conditions must hold in order for the class to be accepted by the OOMatch compiler.

Condition 4.5.1 *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 $m_1 \not\prec_M m_2$, and $m_2 \not\prec_M m_1$, then $\exists M[\theta_3, n, \vec{\phi_1} \sqcap \vec{\phi_2}, \vec{\theta_{T3}}] \in M_C$.*

$\sqcap$	$\alpha = F[\theta_1]$	$\alpha = C[v_1, \theta_1]$	$\alpha = P[\theta_1, n, \vec{P_\alpha}]$
$\beta = F[\theta_2]$	α if $\theta_1 <: \theta_2$ β if $\theta_2 <: \theta_1$ undefined otherwise		
$\beta = C[v_2, \theta_2]$	β if $(\theta_1)v_1 == v_2$ undefined otherwise	α if $v_1 == v_2$ and $\theta_1 <: \theta_2$ β if $v_1 == v_2$ and $\theta_2 <: \theta_1$ undefined otherwise	
$\beta = P[\theta_2, n_2, \vec{P_\beta}]$	β if $\theta_2 <: \theta_1$ $P[\theta_1, n_2, \vec{P_\beta}]$ if $\theta_1 <: \theta_2$ and $\theta_2 \not<: \theta_1$ undefined otherwise	undefined	see Def 4.5.1

Table 4.1: Partial function $\sqcap$ (parameter intersection)

Condition 4.5.2 *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 the static types of the argument to the call are subtypes of the* Java *types of $\vec{p}$, as defined by the type function in Section 4.3.*

Condition 4.5.3 *Completeness: This condition is used to prevent no-such-method errors, unless the programmer overrides it by labelling methods* **inc** *(see Section 3.1). Every method m in a class C that contains patterns must be preferred over another method in C or a superclass of C that contains only Java formal parameters, or m must be labelled* **inc**. *This condition is used to prevent no-such-method errors except where the programmer explicitly allows them.*

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

Definition 4.5.2 *can-both-apply(m_1, m_2) is a predicate on two methods $m_1 = M[\theta_1, n_1, \vec{\alpha}, \vec{\theta}_{T1}]$ and $m_2 = M[\theta_2, n_2, \vec{\beta}, \vec{\theta}_{T2}]$, in the same class or in a subclass or superclass. It is true if and only if all of the following hold:*

- $n_1 = n_2$ *(Same names.)*

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

Condition 4.5.4 *Valid return types: For any pair of methods* m_1, m_2 *such that can-both-apply*(m_1, m_2)*, their return types must be the same.*

Condition 4.5.5 *Valid "throws" clauses: For any pair of methods* m_1, m_2 *such that can-both-apply*(m_1, m_2)*, their throw types must be the same, with one exception. If one of the methods is preferred to the other (without loss of generality assume* $m_1 \prec m_2$*), and* m_1 *and* m_2 *both have only regular Java parameters (no patterns, literal values, or "where" clause), then the throw types for* m_1 *need only be a* subset *of the throw types of* m_2*. (This exception is important for backward compatibility with Java.)*

Condition 4.5.6 *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* $\alpha_i \neq \beta_i$*. (Unless at least one method has a "where" clause, in which case this condition does not apply.)*

Note that the conditions given here apply only to OOMatch code. There are special rules in place when an OOMatch class extends a regular Java class, which are discussed in Section 6.4.

4.6 Absence of runtime ambiguities

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

4.6.1 Undecidable equivalence

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

Definition 4.6.1 *undecidable-equivalence is a predicate on pairs of OOMatch parameters. undecidable-equivalence*(α, β) *is true if and only if* $\alpha = P[\theta_1, n, \vec{\phi_1}]$ *and* $\beta = P[\theta_2, n_2, \vec{\phi_2}]$*, where deconstructor*$(\alpha) \neq$ *deconstructor*(β)*, and either* $\theta_1 <: \theta_2$ *or* $\theta_2 <: \theta_1$*.*

Definition 4.6.2 *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*$(\alpha_i) =$ *deconstructor*(β_i) *and undecidable-equivalence-list*$(\vec{\phi_1}, \vec{\phi_2})$*.*

4.6.2 Common descendent

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

Definition 4.6.3 *common-descendent*(α, β) *is defined for a pair of parameters unrelated by subtyping, i.e. when neither type*$(\alpha) <:$ *type*(β) *nor type*$(\beta) <:$ *type*(α)*. It is true of 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 4.6.4 *common-descendent-list*$(\vec{\alpha}, \vec{\beta})$ *is true if and only if* $|\vec{\alpha}| = |\vec{\beta}|$ *and there exists* i *such that either:*

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

Claim 4.6.1 *Because Java disallows multiple inheritance, common-descendent can only be true if at least one parameter is an interface.*

4.6.3 Deterministic deconstructors

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

4.6.4 Claims of safety

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

Claim 4.6.2 *An ambiguity error cannot occur at runtime unless one of the following conditions is true.*

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

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 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 4.6.1, which is 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 4.5.2, 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 4.6.1 is that m_3 is applicable, and therefore in A. By Lemma 4.6.2 (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 4.6.1 *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 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 [GJSB96, 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 [GJSB96, 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 [GJSB96, 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 [GJSB96, 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, $\theta_2 =$ `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 $\theta_1 =$ NullType. But the only value that is equal to `null` is `null` (see [GJSB96, 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 ([GJSB96, Section 5.6.2]), it is also the case that $==$ holds if r and v are treated as having the type that they are 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 $(\theta_1)r$`==`$(\theta_1)v$, 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 are still 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 the 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 are still 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 impossible 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 impossible 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 β, its static type is either a subtype or supertype of θ_2. Therefore, 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 4.6.1 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 r is admissible to α, its static type is either a subtype or supertype of θ_α. Therefore, 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 4.6.2 *Consider a pair of parameters a and b whose intersection is defined, as $c = a \sqcap b$. If undecidable-equivalence(a, b) is false, then $c \prec_P a$ and $c \prec_P b$.*

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. Without loss of generality, 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 be defined 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. Without loss of generality, 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 5 of $\prec_P$ are applicable and $c \prec_P b$.

5. $a = P[\theta_1, n, \vec{\alpha}], b = C[v, \theta_2]$. Intersection is always undefined in this case, so it does not 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 do not 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 5 of $\prec_P$ are met.

   Without loss of generality, $c \prec_P b$ 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 statically in many common cases.

Chapter 5

Implementation

A prototype implementation for OOMatch has been written with the help of the Polyglot compiler framework [POL], version 1.3.4. We first give a brief overview of Polyglot, then give a general explanation of how OOMatch was implemented. The OOMatch implementation can be downloaded from `http://plg.uwaterloo.ca/~a5richar/oomatch.html`. It has been tested and debugged on small programs, but has not yet been used extensively on a large code base, so it should be considered beta software and not fit for safety-critical applications.

5.1 Overview: how Polyglot works

Polyglot is an extensible compiler framework designed to easily add language features to the Java programming language. The base Polyglot compiler compiles Java code to Java. The intention is for language implementers to add extra classes to extend Polyglot so that it compiles an input language that is slightly different from Java. The output can then be compiled to bytecode with Sun's Java compiler (for example). As long as the input language is sufficiently similar to Java, the code required to extend Polyglot is usually small and can be written quickly compared to writing a compiler for the desired language from scratch.

The Polyglot code should rarely, if ever, need to be modified directly by the user. Instead, Polyglot contains several "hooks" to allow the implementer to customize various parts of the Polyglot compilation process. The most obvious of these is regular class inheritance and overriding, but Polyglot also includes what it calls *extensions* and *delegates*, which are sometimes useful for more complex situations. These are discussed in detail in Section 7.

Like many compilers, Polyglot first performs a parsing phase, during which an abstract syntax tree (AST) is constructed, followed by a number of transformations on the AST. The implementer can customize the provided Java grammar using a special format that allows the specification of insertions, deletions, and modifications to the grammar in a separate file, i.e., without modifying the Java grammar file. The grammar is written in the CUP parser generator format [CUP96], an LALR parser generator for Java.

After parsing, Polyglot performs a series of transformations on the resulting AST using Visitors. [GHJV94] Each visitor, rather than modifying the AST, returns a new AST which becomes the input to the next compiler pass (until the AST is finally output in the last pass). In addition to the AST, Polyglot maintains type information for the input code (such as the classes in the program and what methods they contain), which is built during various compiler passes. Implementers insert compiler passes between Polyglot's passes to do typechecking, type construction, rewriting, or transformation to Java. The passes added for the OOMatch compiler are discussed in Section 5.3. An overview of the passes present in the base Polyglot compiler, after parsing, are as follows:

- build-types: Type information is constructed for various elements of the input code. Most of it is *ambiguous* at this point – that is, any identifiers referenced need not exist, and information about them may be unknown.
- clean-super, clean-sigs: Ambiguities are resolved for classes and method headers – that is, names in them are looked up and, if they can not be found, an error is given.
- add-members: Methods are added to class bodies in the type information.
- disambiguate: All remaining code (i.e., code in method bodies) is disambiguated.
- type-check: Typechecking is performed.
- reach-check, exc-check, etc.: Various static analyses required by Java rules are performed (exception propagation, unreachable code checking, returning from methods, declaration-before-use checking).
- The AST is output to a Java file.

5.2 Overview: OOMatch compiler

The OOMatch compiler extends Polyglot to translate OOMatch to Java. As it is only a prototype compiler, we are not overly concerned with efficiency (either of the output code or of the compiler), but mainly in doing the translation simply. As well, since the compilation is done in two steps (OOMatch to Java, then Java to .class files), there are necessarily efficiency losses compared to a well-written direct translation.

The main translation strategy consists of two steps:

- For each class, build a DAG (directed acyclic graph) of its methods, representing overridden methods. Nodes m and n in the graph, with an edge from m to n, represent that the method n overrides the method m.
- Add *dispatchers* to classes to intercept method calls and determine which method to call. Original methods from the input are renamed to prevent them from being called directly. The dispatchers use the DAG from each class to determine which methods to check for applicability and when to throw an ambiguity error at runtime. A dispatcher also attempts to check methods in superclasses for applicability, if no method in the current class applies.

This process is illustrated at a very high level in Figure 5.1.

In reality, the OOMatch compiler needs to perform many AST transformations in a careful sequence in order for the methods and deconstructors to be able to reference each other and find each other's definitions within the type information maintained for each class. These steps are described in some detail in the following subsection.

5.3 Preprocessing Visitor Passes

After parsing the input code according to the grammar given in Section 4.1, the OOMatch compiler needs to insert several visitor passes between Polyglot's early visitor passes that construct the type information data structures, because assumptions the Polyglot compiler makes for Java code are not true of OOMatch code. A list of all the passes, interspersed with the Polyglot passes, are shown in Figure 5.2, with the new OOMatch passes in bold. A description of the new visitor passes that must take place for OOMatch is as follows:

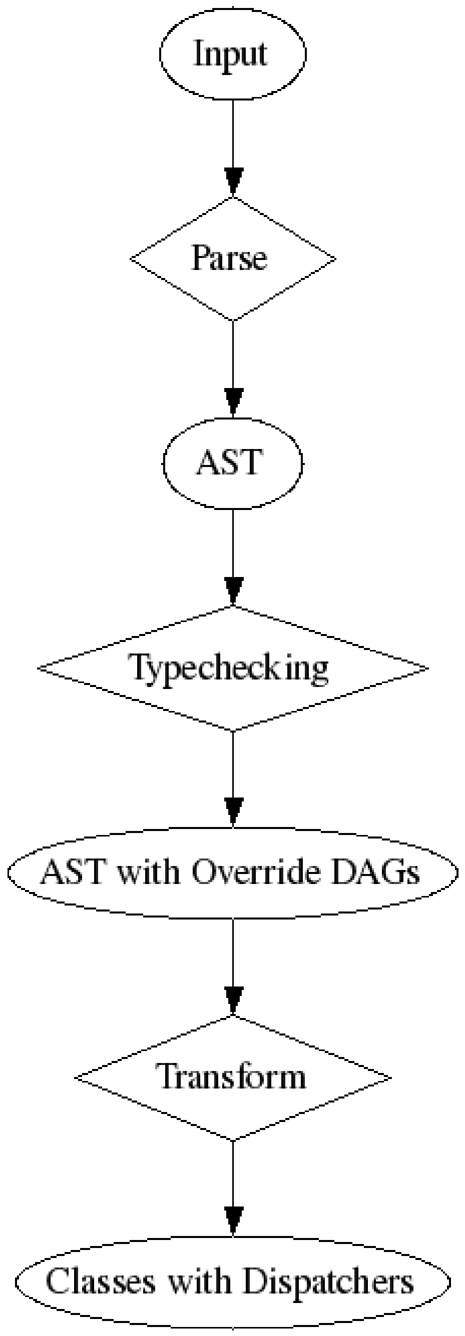

Figure 5.1: OOMatch compiler overview

- **desugar-constructors**
- build-types
- **create-dec-type**
- clean-super
- **create-deconstructor**
- **add-deconstructor**
- **var-decl**
- clean-sigs
- add-members
- **disam-methods**
- **add-methods**
- disambiguate
- **add-manual-children**
- **name-methods**
- **desugar-named-patterns**
- **oom-type-check**
- type-check
- **rename-methods**
- **let-dollars**
- **transform-calls**
- **transform-patterns**
- static analysis checks
- translation

Figure 5.2: OOMatch Visitor Passes

- desugar-constructors: The syntactic sugar for access specifiers in constructors, described in Section 3.2, is desugared as described in that section.
- Polyglot runs its pass to build type information (build-types). During this phase, deconstructor declarations get a special kind of type object built for them so that they can be distinguished from regular methods.
- create-dec-type: During parsing, the section of a pattern before the left parenthesis has been parsed as a simple list of identifiers separated by "."s, without knowledge of what the identifiers represent. This text is now split up into a type and a deconstructor name (the latter possibly being implicit, i.e., being the same as the type name). The type is still ambiguous (so it is as yet unknown what members it contains), and therefore it is as yet unknown whether the deconstructor being referred to exists.
- Polyglot removes ambiguities in the "extends" clause (clean-super), so that super types can be looked up.
- create-deconstructor: Type information for deconstructors is created (corrected, actually, as a shell for the type information is created earlier). It is necessary to do this separately from regular methods because regular methods can reference deconstructors, and they need the correct information to be constructed themselves. This pass simply does what Polyglot would do for regular methods, but only for deconstructors.
- add-deconstructor: Deconstructors are added to the type information objects of the class they are contained in; again, this has to be done separately, so that when type objects for methods are created, they have the deconstructor type information to look up.
- var-decl: The set of variables declared in methods are calculated. For example, the method:

  ```
  void f(Point(int x, int y), Point(0, int z) p) { ... }
  ```

  has four variables declared in it: x, y, z, and p, even though it has only two parameters. Notice that there is a tension between the types and names of the variables declared in the parameters, which the method body needs to see, and the types of the parameters themselves, which clients of the method need to see. For example, the body of `f` sees `f` as a method providing four values x, y, z and p. Places that call the method, however, must see it as a method taking two Points. In the implementation, the AST node contains both the variable declarations and the parameter types, while the type object for a method only contains the parameter types. This causes Polyglot's checking of declaration-before-use to see only the user variables declared in a method, while the checking of method calls sees only the external type of the method (`f(Point, Point)` in this case) - i.e., only the types of the top-level parameters.

- Type information for parameters is partially constructed (clean-sigs). The types of any named variables within the parameters or patterns are looked up.

- Fields are added to classes (add-members). Whereas Polyglot would add both fields and methods to classes during this phase, OOMatch must leave method type construction and insertion until after fields can be referenced, and so overrides this pass to do so. This change is because fields might be referenced in patterns (Section 3.7).

- disam-methods: Type information for method parameters and methods themselves is constructed. For patterns, this involves determining the type of the pattern and binding the appropriate deconstructor to it, as described in Section 4.3.

- add-methods: Methods are added to their classes' type information.

- Regular lookup of names referenced in code occurs (disambiguate).

- add-manual-children: Any manual override relationships (`|` operator) are recorded in the type information.

- name-methods: The output name for user methods is computed and remembered, but not yet changed. This action is done now, rather than during transformation, because later some methods that need renaming are hidden from the type information.

- desugar-named-patterns: The desugaring of named variables (non-linear pattern matching or referring to class variables within patterns), as described in 3.7, is performed.

- oom-type-check: OOMatch typechecking occurs. This step involves checking for ambiguity errors as well as a number of other minor errors, including those described in Section 4.5.2. It also involves building data structures to represent the override relationships among methods. This process is described in detail below in Section 5.4.

- Polyglot performs its typechecking (type-check). Because method type information contains a view of the method as seen by clients, and methods that are too specific to be called directly are removed from the list of methods in the type information for the class (see Section 5.4 below), Polyglot's typechecking catches invalid method calls and duplicate methods as described in Section 4.5.2.

  During this phase, any anonymous classes (classes within a "new" statement) have not yet had the processing above done on them. If one is found, the above passes are run on it at this point.

- rename-methods: The new method names calculated in name-methods are now assigned to the methods. (They were not assigned earlier so that typechecking would work correctly.)

- let-dollars: Examine the body of methods containing let statements (Section 3.12) to determine how to create unique names for variables in the generated code that implements the let statement.

- transform-calls, transform-patterns: Translation of the OOMatch AST to a Java AST occurs. This process is described in detail below in Section 5.5.

- Polyglot runs its semantic analyses required by the Java rules (static analysis checks). The declaration-before-use check catches cases where the "out" parameters of deconstructors are not assigned a value on all paths through the deconstructor.

- Polyglot performs its output of the AST to the output file (translation).

5.4 Typechecking and Building Override DAGs

The typechecking algorithm conceptually builds a set of directed acyclic graphs representing overriding relationships. Note that a method may override multiple methods, and hence, have multiple parent nodes - i.e., the graphs are DAGs and not necessarily trees. Concretely, these DAGs are stored as "children" lists in the method's type information. That is, each method stores a list of methods that directly override it. When it is found that a method A overrides a method B, A then becomes a child node of B in B's graph, with the parent node B pointing to A. The conditions from Section 4.5.2 are also checked in the typechecking pass, either during or apart from the building of the DAGs.

The specific algorithm to build these structures and do the check for ambiguities is shown in Algorithms 1 and 2. The algorithm starts off with an empty DAG structure (line 1). For each method m (line 2), it takes the top-level methods in the DAG (root nodes) and tries to find a place to insert m, returning a new DAG structure as a result (line 3).

```
Input: A class C to typecheck
Output: Nothing; C is modified to have some of its methods hidden from
        clients, and its methods will contain children lists; or, an error is
        thrown if there is an error in C.
1 L = empty list;
2 foreach method m in C do
3     (L, wasAdded) = addToMethods(m, L);
4     if not wasAdded then
5         add m to L;
6     end
7 end
8 Remove all methods from C that have patterns and that are not top-level (i.e.
  are not in L);
```

Algorithm 1: typecheck

If m overrides one of the top-level methods, it is recursively placed somewhere among the children of that method (lines 7 to 14 of addToMethods). If one of the top-level methods overrides m, that method is placed in m's children, and m becomes a top-level method (lines 4 to 6 of addToMethods). In either case, the process continues for all the top-level methods, since a method might have multiple parents or multiple children.

We also do the check for ambiguities from Section 4.5.2 directly in the algorithm (lines 15 to 17 of addToMethods). This location is a convenient (and efficient) place to do the check because it only needs to be done when neither method is preferred to the other.

At the end, if the method has not been placed anywhere, it becomes a new top-level method by itself (lines 4 to 6 of typecheck).

After all methods have been processed, methods that have a parent in the DAG and that have patterns are removed from the class's type information (line 8 of

```
Input: A method $m$ to add to a DAG structure $L$
Output: A pair. The first element is the new DAG structure, possibly with
        $m$ added. The second is a flag saying whether $m$ was added
        somewhere below any of the roots of $L$.
$L'$ = empty list;
$wasAdded$ = false ;
foreach $m'$ in $L$ do
    if $m'$ overrides $m$ (as determined by the preference operator $\prec_M$ from
    Section 4.4.2) then
        add $m'$ to children of $m$;
    else
        if $m$ overrides $m'$ then
            ($newChildren$, $wasAdded'$) = addToMethods ($m$, $m'$.children());
            set $m'$.children to $newChildren$;
            if not $wasAdded'$ then
                add $m$ to the children of $m'$;
            end
            $wasAdded$ = true ;
        else
            if intersection of $m$ and $m'$ is defined and does not exist as a
            method in $C$ (intersection determined by the $\sqcap$ operator from
            Section 4.5.1) then
                throw ambiguity error;
            end
        end
        add $m'$ to $L'$;
    end
end
return ($L'$, $wasAdded$);
```

Algorithm 2: addToMethods

typecheck). This removal is to prevent regular Java typechecking from finding duplicate definitions (as the Java type of a pattern may be the same as that of a regular formal parameter), and to enable it to typecheck method calls normally.

To further help understand the algorithm, an example is given in Appendix A.

Note that although in algorithm 1, the list of top-level methods L is a flat list, in the implementation we actually split up L to group together methods of the same name, using a hash table. Use of this technique means that each method is only compared against other methods of the same name, which is much more efficient most of the time.

In the worst case, the running time of Algorithm 1 is $O(n^3)$, for a class with n methods. This worst case occurs when every method is checked against every other method, and each pair of methods has a non-empty intersection, so the compiler has to do a linear search for the existence of that intersection. However, in practice, the algorithm is rarely this bad, because a method is only compared against methods of the same name. The algorithm does significantly worse when the program contains a class with a large group of overloaded or overridden methods, but it is rare that there is a large enough group to significantly affect performance.

5.5 Transformation

Transformation of the typechecked OOMatch AST to Java takes place in several steps, described in the next several subsections.

5.5.1 Rename methods

The first step is to find a new name for methods; in the output Java code, they are renamed to this new name. Methods can not necessarily have the same name the user gave them, because if they did, there might be duplicate definitions in the output Java code. For example, these two methods:

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

would result in a duplicate definition if they are not each given a unique name. The name mangling scheme is described in Section 5.5.5.

In this step, each user method also receives a unique integer identifier, which is used later to remember whether that method applied.

5.5.2 Add dispatchers

The bodies of user methods (methods in the input program) remain the same, except that user methods are all renamed so they can not be called directly. Instead, special methods called *dispatchers* are inserted into classes by the transformation phase. Method calls in the final Java code calls these dispatchers, which in turn select the appropriate user method to call.

Many dispatchers of a given name, each handling different parameter types, may be added to each class. Each dispatcher completely determines the method to call, by itself - dispatchers do not call each other. A dispatcher uses the override relationships represented in the DAG structures, which were created during typechecking, to determine the method to call. If the dispatcher cannot find an applicable method in the DAG, it checks methods in the superclass. If an applicable method cannot be found in any superclass, it throws a no-such-method error.

An outline of the algorithm that a dispatcher uses to determine the method to call is given in Algorithm 3. This outline is an abstract algorithm only; the dispatchers that actually get created do not know anything about a DAG, because there is no DAG at run-time.

The basic idea of the algorithm is to take all the *leaves* of the DAG - methods which are not overridden - and check each for applicability. When a method applies, it is remembered. If a method is found to apply, and one has already applied, there is an ambiguity error. This process is then repeated for all the new leaves of the smaller DAG. A method is only checked for applicability if none of its descendents have applied (which have all been checked by the time the method itself is checked). This bottom-up traversal ensures that the most specific methods are always chosen first.

Note that a dispatcher is created for each top-level method (root node) of the DAG, which ensures all method calls find an appropriate dispatcher. The parameters the dispatcher is given match the *Java* types of the parameters of the top-level method for which it is being created. (Since a method must override a method with regular parameters unless it is labelled `inc` (see Section 3.1), this usually means that the dispatcher receives the same parameter types as its corresponding method.)

Ensuring Correct Semantics Among a Class Hierarchy

The approach described above is sufficient to correctly determine which method to call when all the method cases are within the same class. Ensuring that the correct

```
$C$ = the class that the dispatcher is being created for;
repeat
    while DAG of $C$ is not empty do
        method = 0;
        leaves = all the leaves of the DAG;
        foreach $m$ in leaves do
            if method $\notin$ $m$'s descendents and $m$ applies then
                if method = 0 then
                    method = $m$;
                else
                    throw ambiguity error;
                end
            end
        end
    end
    if not method = 0 then
        call method;
        return ;
    end
    $C$ = superclass of $C$;
until $C$ is Object ;
throw no-such-method error;
```
Algorithm 3: Dispatcher pseudocode

method gets selected among a class hierarchy - and that methods in subclasses are always preferred to methods in superclasses - is more complex. The difficulty is that Java does not consider methods to override other methods unless their parameter types are exactly the same. For example:

```
class A {
    void f(Collection c) { ... }
}

class B extends A {
    void f(List l) { ... }
}
```

The programmer should expect `B.f` to override `A.f`. But the dispatcher for `B.f` - which also has a parameter of type `List` - will not catch the following call:

```
A a = new B();
a.f(new List());
```

According to Java semantics, the dispatcher for `A.f` will be called instead. The problem cannot be solved by adding code to `A` to try to dispatch to subclasses, because (to give only one problem with doing so) `A` might be in a library and unable to be modified or to know what its potential subclasses might be.

To solve the problem, we add extra dispatchers to the subclass with the exact parameters of dispatchers in the superclass. Any method in a superclass with a dispatcher (or that will get a dispatcher) that might apply at the same time as a method in the current class, gets pulled down as a dispatcher in the current class. These dispatchers, like the regular ones, check all methods in the class that might apply to their parameters, most specific ones first, and then check superclasses if no applicable method can be found.

Creating the Dispatchers

A simplified (not exactly accurate) algorithm to create the dispatchers is shown in Algorithm 4 and the following algorithms. Lines 3 to 4 of createDispatchers show the creation of dispatchers for methods in the current class. Lines 5 to 8 show the creation of dispatchers that are pulled down from methods in the superclass, as described above.

Algorithm 5 (createDispatcherFor) gives an overview of the creation of a single dispatcher. As the dispatcher checks methods for applicability, it keeps track of any method that has previously applied with the use of a single integer variable, (lines 3 to 4). A value of 0 means no method has yet applied; when a method applies, this variable is set to the unique integer ID of that method. Whenever a method applies and this integer already contained a non-zero value, it is an ambiguity, and an error is thrown.

A dispatcher has to check the methods in the current class, followed by the methods in each superclass in turn, for applicability. (Lines 5 to 8). At the end of the checks it adds a throw statement for no-such-method errors (line 9). Recall that, since a method with patterns must override a regular Java method unless the former is labelled `inc` (see Section 3.1), this throw statement can only be reached for `inc` methods.

Also note that dispatchers receive parameters with the types corresponding to the *Java* types of the OOMatch parameters from the method that the dispatcher is being created for. (Lines 1 to 2). Since this might cause the same dispatcher to be created for multiple different methods (in the case of `inc` methods), there is a check done on Line 7 of createDispatchers which prevents duplicate dispatchers from being added to the class.

Algorithm 6 (dispatchMethods) shows the creation of the code to add to a dispatcher to attempt to dispatch to a *single* class. Recall that each class now contains a DAG representing which methods override which. The algorithm uses (a copy of) this DAG to decide which methods to dispatch to first. For each iteration of the outer loop (line 6), it removes the *leaves* from the DAG (i.e., the nodes without any edges to other nodes), and produces code to dispatch to each of the methods represented by these leaves. It then iteratively removes the next group of leaves from the DAG, repeating the process until the DAG is empty. Because checking each method is only done if none of the method's children have applied yet, this ensures that children are always checked before their parents, and hence that the most specific method is always chosen first.

The steps to create the code to attempt to dispatch to a particular method is shown on lines 10 to 23. Basically, the steps that are done are as follows.

- Any deconstructors that need to be called to determine applicability are called. (Line 15.)
- A guard checking that no child of the method has yet applied (the method is skipped if one has) is created. (Lines 19 and 21.)

- The condition to check for applicability for the method is created. (Line 21.)
- A check that no other method has applied, throwing an ambiguity error if one has applied, is created. (Line 20.)
- The method is called. (Line 22.) Note that calls to the method are done after all methods in the class are checked for applicability (in case there was an ambiguity error). The call is done in a switch statement (Lines 9 and 22) that uses the methodChosen integer to remember which method to call.

The details of checking a method for applicability, as well as creating calls to deconstructors and the arguments to pass to the method, are done separately on lines 13 to 18 of dispatchMethods, and in Algorithm 7 (dispatch). Basically, for each parameter of the potentially applicable method, three main things need to be calculated:

- The set of calls to the deconstructors in the method's parameters. These need to be calculated recursively, i.e. subpatterns in the method take the result from the outer deconstructor and call another deconstructor on one of its components.
- The boolean condition that determines whether the method applies must be built. This consists of matching the parameters of the dispatcher against the patterns in the method.
- Any arguments that will need to be passed to the method. For each identifier declared within the method's parameters, a corresponding argument will need to be passed if the method applies.

Algorithm 7 basically implements the *admissible* predicate given in Section 4.4.1. The code pieces that need to be calculated for the parameter depend on the type of parameter.

1. For patterns (lines 1 to 12), we first need to check that the argument is a subtype of the pattern type (lines 2 to 3), to implement multimethod behavior. The call to the deconstructor is then performed (lines 4 to 5). Note that the call is only performed, and the method only matched, if the argument is non-null, because null never matches any pattern.

   The subpatterns are then matched for applicability on lines 7 to 12.

2. For literal parameters (lines 13 to 18), we simply need to add a check for whether the argument is equal to the literal. This involves calling `.equals` for string literals, and using `==` for other literals.

3. For regular Java formals (lines 19 to 26), we also need to do the subtype check for multimethods. For primitive types, this involves a check that casting the value to the desired type does not affect it (line 21). We also have to add the argument to the arguments that will be passed along to the user method, making sure to cast it to the formal's type first (line 25). This cast is only done once the instanceof succeeds, so it should never crash.

```
Input: A class C to add dispatchers to
Output: No output; adds new dispatchers to C
1  newDispatchers = empty list;
2  foreach top-level method m in C do
3      n = createDispatcherFor (m, C);
4      add n to newDispatchers, unless it is already there;
5      foreach method m′ in an OOMatch superclass of C (with some
       exceptions), such that the intersection of m and m′ is not null do
6          n = createDispatcherFor (m′, C);
7          add n to newDispatchers, unless it is already there;
8      end
9  end
10 add all newDispatchers to C;
```

Algorithm 4: createDispatchers

Alternate considered approaches

The algorithm could certainly benefit from speed improvements of the output code. One major drawback of our algorithm is that every method must receive a dispatcher, even if pattern matching is not present anywhere in the class. To understand why this is so, consider the following code:

```
class A {
    void f(Point) { ... } //Suppose this method does
        //not get a dispatcher
}
```

```
Input: A method $m$ to create a dispatcher for, and a class $C$ to put it in
Output: The new dispatcher
1  $newParams$ = the Java types of the parameters of $m$ (according to the type
   function defined in Section 4.3);
2  $c$ = new method with parameters equal to $newParams$;
3  $methodChosen$ = an integer variable to keep track of which method was
   chosen;
4  add to the body of $c$ a declaration of methodChosen;
5  while $C$ != Object and $C$ is an OOMatch class (as opposed to a Java
   class) do
6      Add to the body of $c$ the result of dispatchMethods ($m$, $C$,
       $newParams$, $methodChosen$);
7      $C$ = super class of $C$;
8  end
9  add a throw statement as the body of $c$ to throw a no-such-method error;
10 return $c$;
```

Algorithm 5: createDispatcherFor

```
class B extends A {
    void f(Point(int x, int y)) { ... }  //Gets a dispatcher f(Point)
}

class C : B {
    void f(CoorPoint(0, 0));
}
```

The dispatcher for `C.f` might check whether `B.f` applies, and find that it does not (because `Point`'s deconstructor returns false). It then finds that `A.f` does apply. But, in Java, there is no way to call `A.f` from `C`; one can only call `super.f`. But once `B` gets a dispatcher, it will intercept the super call, and check `B.f` for applicability a second time (possibly resulting in a match). This violates our semantics that a pattern is only checked for applicability once for a given call.

The problem would be easy to solve if the the language provided a way to call `A.f` directly (C++ allows this, for example). Instead, we decided to create a dispatcher for every method, so that the original user method always has a unique name, and hence can always be called directly from any subclass.

Input: A method m to create a dispatcher for
A class C that contains the list of methods we're trying to dispatch to
$newParams$, the parameters of a new dispatcher method being created
$methodChosen$, an integer variable that keeps track of which method was chosen
Output: Code to place in the new dispatcher to attempt to dispatch to all possibly applicable methods in C

```
M = The top-level methods in C (with some exceptions);
Remove all methods from M that cannot simultaneously apply with m;
(determined as in Section 4.5.2)
d = A DAG formed from each element of L as a root node, and each having
  child nodes being the same as the DAG structures calculated during
  typechecking (Section 5.4).;
stmts = empty list of Java statements;
while d is not empty do
    L = all the leaf nodes in d;
    remove all the leaf nodes from d;
    s = new empty switch statement;
    foreach method m in L do
        cond = true;
        args = empty list;
        foreach parameter p in newParams, p′ in m do
            (decls, cond′, args′) = dispatch (p′, p);
            add decls to stmts;
            cond = cond and cond′;
            add args′ to args;
        end
        ids = a list of the method IDs given to all methods (directly or
          indirectly) preferred over m;
        err = an "if" checking whether methodChosen != 0, with a body
          that throws an ambiguity error;
        add to stmts an "if" checking whether (methodChosen != any
          elements of ids) && cond, with a body containing err and an
          assignment of methodChosen to the ID of m;
        add to s a case that calls m with arguments args;
    end
end
add s to stmts;
return stmts;
```

Algorithm 6: dispatchMethods

Input: A parameter p' being matched (from the child method)
A variable v being checked for a match (from the parent method)
Output: A triple $(decls, cond, args)$ where:
decls is a list of variable declarations and calls to deconstructors for the dispatch code
cond is a condition to check whether the method applies
args is a list of the arguments to pass to the child method

```
if $p' = P[T, n, \vec{p}]$ then
    $cond$ = "v instanceof T";
    add "(T)v" to $args$;
    add the declaration of a new variable to decls - call it $r$ - to hold the
    result of the deconstructor call;
    add "if (v != null) r = v.n()" to decls - (i.e. a call to the
    deconstructor);
    add "&& v != null" to $cond$;
    foreach $p$ in $\vec{p}$ do
        $(decls', cond', args')$ = dispatch ($p$, $v'$);
        add $decls'$ to $decls$;
        $cond$ = "cond && cond'";
        add $args'$ to $args$;
    end
else if $p' = C[val, T]$ then
    if T == String then
        $cond$ = "val.equals(v)";
    else
        $cond$ = "val == v";
    end
else if $p' = F[T]$ then
    if $T$ is a primitive type then
        $cond$ = "(T)v == v";
    else
        $cond$ = "v instanceof T";
    end
    add "(T)v" to $args$;
end
return $(decls, cond, args)$;
```

Algorithm 7: dispatch

Another approach would have been to have dispatchers always call the most general super dispatcher it can find instead of checking methods in the superclasses for applicability itself. Under this approach, it would get complicated to ensure that the static type information of the parameters is not lost (to preserve the semantics for null arguments described in section 4.4.1). Further, though this optimization appears to improve speed of the output code, the extra dispatchers can actually be optimized away by a good optimizer, and hence speed is not necessarily improved over properly optimized code. An optimizer can notice that the dispatcher contains a condition that is always calling the user method, and hence remove everything but the call, and then inline the method body, and finally notice that the user method is never being called, and remove it. Hence, since this implementation is simplest, we decided to rely on an optimizer to remove the overhead. To aid the optimizer in doing so, we label all concrete renamed user methods "final", because the new names never conflict with each other and never override each other.

5.5.3 Transform deconstructors

The next step is to transform deconstructors, which is necessary since their "out" parameters are not a feature of Java. This involves 3 main transformations:

- Deconstructors are transformed to a method with no parameters, and the "out" parameters given to it by the user become local variable declarations at the start of the method.
- The return value of the method becomes `Object[]`.
- Any return statements in the deconstructor are translated to a condition that checks the boolean value that was previously returned. If false, null is returned in the new method; if true, a new `Object[]` is returned, initialized with the former "out" parameters. Primitive values are boxed into their corresponding class (for example, for an `int` variable `x`, `new Integer(x)` would be put in the `Object[]`). They are later unboxed in the dispatch code, by calling, for example, `Integer.intValue()` (this occurs on line 10 of Algorithm 7).

Also, note that deconstructors must get a new unique name, because if there are overloaded deconstructors, they would otherwise all get transformed to methods of the same name with no parameters, which would cause a duplicate definition in Java. Note, too, that there is no danger of deconstructors having the same name as the class they are contained in, because methods are allowed to have the name of their container class in Java.

5.5.4 Passing null and primitive values

As mentioned in Section 4.4.1, a null value should match a parameter of regular Java reference type only if the static type of the argument passed to the method is a subtype of the parameter's type. Likewise, a method taking a primitive type parameter should only match if the static type of the argument is method invocation convertible to the parameter's type. While these semantics make the most sense to the user, it turns out that they require a bit extra to implement in the context of our algorithm given above, because this static type information is not automatically available to the dispatcher method at run-time.

For example, consider this code, where `B extends A`:

```
class C {
    void f(A a1, A a2) { ... }
    void f(B b, A(0)) { ... }
}
...
A a = null;
f(a, new A(0));
```

The compile-time method selected for the call is the first `f`. But, even if the deconstructor for the second parameter of the second method applies, it does not make sense to call it, because the null A is not a B. On the other hand, if the user changes the call to the following:

```
f((B)null, new A(0));
```

then it should be interpreted as an intention to possibly call the second version of `f`. But the problem is that the dispatcher called has a parameter of type `A`, and cannot distinguish from that parameter being a plain `null` and a `(B)null`. A similar problem happens with primitive types – for example, if the code were changed to:

```
class C {
    void f(int x, A a2) { ... }
    void f(short x, A(0)) { ... }
}
...
f((short)0, new A(0));
```

Both these problems were solved in our implementation by adding extra parameters of type `Class` to every dispatcher method representing the static type information of the other parameters, and extra arguments to every method call that calls an OOMatch method (as opposed to one in a Java library, such as the Java standard library). For example, the call:

```
f((B)null, new A(0));
```

would be translated to:

```
f((B)null, new A(0), B.class, A.class);
```

While this may appear to cause excessive overhead, it is conceivable that an optimizer with access to the whole program could remove the extra parameters in the (many) cases where they are never used. This removal would in addition be a generally useful optimization. Hence, we chose to avoid over-complicating the implementation with optimizations and defer this task to a separate optimizer, where speed is important.

5.5.5 Name Mangling

Most of the compiler-added code is in the bodies of dispatchers, where there is no risk of name clashing, since all this code is written by the compiler. The main difficulty is that all deconstructors and user methods must be given a new name, and these must each be guaranteed to be unique among themselves and with the compiler-added dispatchers.

To ensure this, we examine the names of all the methods of the class containing the method to be renamed, and its superclasses, and find the one with the maximum number of dollar signs contained in it. Then we add an extra dollar sign to this amount, and add it to the end of the name; this cannot, then, conflict with any of the new dispatchers. Finally, we give each method a unique integer ID (unique from methods in the superclasses as well), and add this to the end of the name. The ID ensures that each user method has a different name.

Note that name mangling causes some minor difficulties for incremental compilation. The notable case is that if there are overloaded or overridden deconstructors, and their lexical order is changed within a class, then any classes that use those deconstructors need to be recompiled, because the names of the deconstructors might have switched. (Note that this is not an issue for method calls, because method calls are not renamed.)

```
static int f(null)
{ ... }
static int f(Point(int x, int y) p)
{ ... }
static int f(Point p)
{ ... }
static int f(Point(0, 0))
{ ... }
static int f(CoorPoint(0, 0))
{ ... }
```

Figure 5.3: Code for example transformation

5.5.6 Example Transformation

An example of part of the generated code is given in Appendix C. It is the dispatcher created for the set of methods shown in Figure 5.3. It is hopefully helpful in understanding how the transformation algorithm works. It is manually commented with an explanation of what each piece of generated code is doing.

Chapter 6

Limitations

The language and its implementation, as they currently exist, have several lamentable limitations, which are as follows. The limitation on error messages (Section 6.3) is a limitation on the OOMatch compiler; the others are limitations on the language specification itself.

6.1 Patterns in Constructors

Constructors may not contain patterns in OOMatch, as it now exists. Conceptually it appears that they should; however, allowing this turns out to be very difficult to implement in Polyglot. The culprit is the (rather restrictive) Java rule that calls to super constructors may only appear as the first statement in a constructor. As has been mentioned in Section 5.5.1, methods are renamed and dispatchers receive the name that the method originally had. But renaming a constructor causes it to cease being a constructor, so this can not be done quite the same for constructors. A simple solution would be to convert constructors in OOMatch code to void methods in the Java output; but, if the constructor had a super call in the input, Java would give an error when compiling the output code, as that super call would now be in a void method.

The problem could be circumvented if OOMatch were compiled directly to bytecode, as bytecode does not have that restriction. This compilation could possibly be done using the Soot Compiler Optimization Framework [Soo], which has Polyglot built into it. However, such a transformation could potentially be a fairly major undertaking, and since patterns in constructors are likely relatively rare, it was decided to keep the prototype implementation simple and leave this for future work.

Several other possible ways to transform constructors were investigated, but all turned out to raise problems of their own. The first possibility is to create a special constructor just for calling super, which takes the formal parameters declared in the user's constructor as an `Object[]`. Calls to "super" could then be transformed into calls to "this" on that new constructor. But it was then discovered that calls to "this" must also be the first statement in a constructor, which brings us back where we started.

Another failed attempt involved transforming calls to the constructor (i.e. the calls to "new") rather than the constructor itself. Rather than calling new, we would call a dispatcher (which has received a different name) that returns an object of the type of the "new". This approach would not work because there might be "super" or "this" calls referencing the constructor, in addition to "new" calls. Unlike the "new" calls, these "super" and "this" calls cannot be replaced with a call to a dispatcher returning an object, because "super" and "this" calls are not expressions; they expect to create the current object. Since a dispatcher can not create the current object without itself calling "super" and "this", we have the same problem as before.

Because it was originally planned to allow constructors to have patterns, the implementation contains code to handle them, but it has been disabled.

6.2 Abstract Deconstructors

It is natural for a programmer to want to give abstract deconstructors to abstract classes, or put them in interfaces, to force implementations of the abstract class or interface to provide a particular deconstructor. However, this feature would potentially complicate the implementation greatly, and so has been left for future work.

The main difficulty lies in multiple inheritance (of interfaces). Suppose there are two interfaces that declare the same deconstructor:

```
interface A {
    deconstructor d(int x);
}
```

```
interface B {
    deconstructor d(int x);
}

class C implements A, B { ... }
```

The problem is that `A.d` would need to be renamed (to avoid conflict with other deconstructors named d), and so would `B.d`. But unless they receive the same name, the deconstructor in `C` which has the signature `d(int)` cannot receive a name that matches both of them, and hence, has no way to be called correctly.

Note that the problem cannot be circumvented by simply omitting deconstructor declarations from interfaces in the output Java code. The dispatchers still need to be able to call the deconstructor, but in general, they do not know about all of the classes that implement the interface. Hence, they must call some method declared in the interface that represents what was originally the deconstructor.

The problem could be partially solved if there were a way to give a unique name to any deconstructor with a certain signature. However, there would still be problems if OOMatch uses Java code that is not transformed (and hence is not renamed). If the class `C` happened to also implement an interface written in Java that contains a method that has the same name as the unique name for `deconstructor d(int)`, then there would have to be a name clash.

Due to these and other complexities, this feature has been left as future work. It should be noted that abstract classes can still contain *concrete* deconstructors, which could return, for example, the class's "getter" methods in its "out" parameters. In this way, abstract representations can still be deconstructed independent of their implementation. Alternatively, a static deconstructor (Section 3.13) can be defined in a separate class to deconstruct abstract classes or interfaces.

6.3 Error Messages

Exceptions in Java, when they occur at runtime, display source files and line numbers of the Java code they were compiled from. As a result, in our OOMatch implementation, the user sees the line numbers of the intermediate Java file when the program crashes. It could be argued that this is not a big issue, since programs are not supposed to crash, but it still slows down debugging as the programmer who wrote the OOMatch code may have to look in the OOMatch output and find the corresponding file in their program.

One way this problem could be solved is to include comments containing line numbers in the Java output, as is done in yacc [Joh79]. A special overhead process would then have to run the program, catch all exceptions, and look up the real line number from the line number given in the exception. Alternatively, perhaps throw statements could be transformed to a special call which replaces the line number of the throw statement with the fixed line number that it appears at. Either way, this problem would likely require a fairly complex solution, and since it is a general problem not really relevant to OOMatch directly, we have left is as future work.

Ideally, a language like Java, which prints line numbers from the code, should also provide a way to override those line numbers, e.g., by allowing special comments or annotations to be placed beside potential exception locations, and printing those instead.

6.4 Extending Java Classes

Java code can be used from within OOMatch, and Java classes can even be extended by OOMatch classes. However, when an OOMatch class extends a Java class, it may not create a method with the same name as one in the Java class, unless it contains the exact same parameter types as the super method (or as *some* Java super method).

To understand why this restriction is needed, suppose it were not in place. Suppose there were the following Java class:

```
class JavaClass {
    void f(Shape s) {}
}
```

Now suppose an OOMatch class wants to extend this class as follows:

```
class OOMClass {
    void f(Circle c) {}
}
```

The problem is that the method `OOMClass.f` is translated so that the method itself is renamed to not override anything, and receives a dispatcher with signature `f(Circle, Class)` (due to the static type information described in Section 5.5.4).

If `JavaClass` is an OOMatch class, this would not be a problem, because then its `f` method would also be translated the same way. But because Java methods cannot be translated, the overriding relation would be broken. But if the dispatcher for `OOMClass.f` does not receive the extra parameter, then it does not know the static type of its argument, and cannot correctly implement the required semantics for null described in Section 4.4.1.

Therefore, while `OOMClass` can still override the `f` in `JavaClass` by creating another method with a `Shape` parameter, the attempt at multimethod overriding shown above is disallowed. (The problem can be circumvented, of course, by migrating the legacy Java code to OOMatch.)

Note that when overriding a Java method with an OOMatch method in this fashion, the OOMatch method is not transformed, but is left as-is. Also, the rules for the return and throws clauses given in Section 4.5.2 do not apply to these methods. The Java rules for return and throw types are used instead. These turn out to be the same as the OOMatch rules for the pair of methods with the same parameters, but if there are other overloaded methods in the Java class, the OOMatch method need not restrict its return and throw types to that method.

Chapter 7

Use Cases

Pattern matching as dispatch can improve code quality to some degree in a wide range of application domains, but it becomes really useful when there is a large class hierarchy and a set of rules for different parts of that hierarchy. This situation occurs with the AST of a compiler. To illustrate this, we show two use cases for OOMatch.

The first use case describes improving Polyglot using OOMatch, by eliminating the need for its delegates and extensions. We have not implemented this improvement, however. The second use case describes converting parts of the Soot Java Optimization Framework [Soo] to use OOMatch. A large Soot class has been transformed as an example of how Soot could be much cleaner with pattern matching as dispatch, and the results of this experiment are given. In the Java version, the class traverses an AST with a Visitor and does various checks for nodes that have a certain form. In the OOMatch version, the need for visitors (which is the source of a lot of boilerplate code within Soot) has been completely eliminated and replaced by a single method call. The code has been made significantly more readable as OOMatch patterns, and the amount of explicit casting and run-time type checks has been drastically reduced. Specifically, the OOMatch version contains only 35 uses of `instanceof`, while the Java version contains 121. The pattern matching has been done without modifying the AST class hierarchy, by using static deconstructors (Section 3.13).

7.1 Polyglot

We have discovered a major way in which the same Polyglot [POL] framework used to implement OOMatch could be simplified if it were implemented in OOMatch instead of Java. Polyglot encounters difficulties as an extensible compiler implemented in a language without multiple inheritance. It has to deal with the expression problem [OZ05], which [POL] calls the mixin problem - nodes in the AST of the new language need to receive new operations both from nodes higher up it their own AST and from the corresponding nodes in the original Java AST.

For example, here is a subset of the Java AST in Polyglot:

```
class Expr { ... }
class Cast extends Expr { ... }
class Binary extends Expr { ... }
```

In the most extreme case, an extension to Java (call it X) has to improve on all the nodes in the AST, and so would need a new node class from each type - `XExpr`, `XCast`, and `XBinary`. The problem is that `XCast`, for example, needs the new features of `XExpr`, as well as the features from the Java AST class `Cast`. Since Java lacks multiple inheritance, it can not simply extend both of them.

Polyglot works around the mixin problem by adding to each node an *extension* and a *delegate*. An extension contains new methods that the programmer wants to add to AST nodes, while a delegate is meant to contain methods that override existing methods in the AST node. These extensions and delegates can be added to any AST nodes. However, this approach means that factory methods are now needed to create the AST nodes and give them the proper delegate and extension, and that methods must be called through the appropriate delegate and extension field of the class, and generally they add extra complexity that, ideally, should not be needed.

In OOMatch, the problem would be simple to solve by creating a class with multiple cases of the function needed to process the AST. For example, where the typecheck algorithm is needed, Polyglot could provide a class containing multiple cases of the algorithm, an outline of which could look like this:

```
class TypeChecker {
    void typeCheck(Expr e) { ... }
    void typeCheck(Cast c) { ... }
    void typeCheck(Binary b) { ... }
    ...
}
```

The first version of `typeCheck` would be overridden by the other two.

If a language extension needs to provide a new typechecking algorithm, it could simply extend `TypeChecker` and override the appropriate method(s).

While there are more fundamental ways to solve the mixin problem here (such as introducing multiple inheritance or Nested Inheritance [NCM04]), the OOMatch solution has the added benefit of allowing the sub-nodes of an AST node to be extracted directly in the parameter list. Visitors frequently do this extraction manually already. For example, one could write something like:

```
void typeCheck(Cast(TypeNode t, Expr e))
    throws SemanticException
{
    if (!e.type().canConvertTo(t.type()))
        throw new SemanticException(...);
}
```

7.2 Soot

Soot is a framework to optimize Java bytecode. Soot translates bytecode to an intermediate representation called *Jimple* to do transformations, and the Jimple representation has its own Abstract Syntax Tree. We have taken some code that processes that AST, found in the file `ConstraintChecker.java` which is in the directory `src/soot/jimple/toolkits/typing/integer/` of the Soot source tree, version 2.2.4, and rewritten it in OOMatch.

The Jimple AST has associated with it a Switch (Visitor) interface to allow traversal. The Switch has a separate method for each node type to implement double dispatch, and looks something like:

```
public abstract class AbstractStmtSwitch implements StmtSwitch
{
    Object result;
    public void caseBreakpointStmt(BreakpointStmt stmt)
    {
        defaultCase(stmt);
    }

    public void caseInvokeStmt(InvokeStmt stmt)
    {
        defaultCase(stmt);
    }
    ...
    public void defaultCase(Object obj)
    {
    }
}
```

Every case contains a call to the default case, which is clearly very verbose. The `StmtSwitch` interface contains similar verbosity. The `ConstraintChecker` class that we have transformed originally extended `AbstractStmtSwitch` and overrode each method in the switch to implement the traversal. The `ConstraintChecker` class itself then calls `apply` to begin traversal. In the OOMatch version, since OOMatch supports multimethods, we simply remove `ConstraintChecker`'s dependence on the Switch and call the method directly in place of the call to `apply`. If this approach were taken throughout Soot, the Switch classes could be eliminated altogether.

The code in `ConstraintChecker` also contains a lot of pattern matching and casting of subcomponents. Static deconstructors (Section 3.13) were used to deconstruct the Jimple AST and do OOMatch pattern matching instead. Following are some examples comparing both versions of the code.

```
    //Java
    if(l instanceof ArrayRef)
    {
     ArrayRef ref = (ArrayRef) l;
     Type baset = ((Local) ref.getBase()).getType();
     if(!(baset instanceof NullType))
     {
         ArrayType base = (ArrayType) baset;
            Value index = ref.getIndex();
```

```
//OOMatch
public TypeNode left(d.ArrayRef
    (ArrayType base, Value index) ref, AssignStmt stmt)
{

...

//Java
//The following pattern occurs many times in the Java version
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), AssignStmt stmt)
{
    return ClassHierarchy.v().typeNode(t);
}
```

Overall, the original version of ConstraintChecker.java contained 1221 lines of code (simply counting the number of lines in the file), and the new version contains 948, not counting the removed dependency on the Switch classes. It should be noted that about 100 of those lines are deconstructor definitions, which could be reused elsewhere throughout Soot on the Jimple AST. It should also be noted that these numbers are too optimistic, because the original code contained a lot of duplication, some of which was factored out in the process. Overall, it could be estimated that the code saved in a tree traversal of that size would approximately balance the extra code needed for deconstructors, so that if many such traversals on the same AST were converted to OOMatch, there would be a significant reduction in code size.

More important than the reduction in code size is the reduction in casts and run-time type information (`instanceof`s). This reduction was roughly quantified by counting the number of `instanceof`s in each version of the class. The original Java version had 121 `instanceof` uses, while the new OOMatch version has only 35. Further, many of the remaining `instanceof` uses are due to testing whether a variable is one of a set of types. For example, the following test appears in ConstraintChecker:

```
if((be instanceof AddExpr) ||
   (be instanceof SubExpr) ||
   (be instanceof MulExpr) ||
   (be instanceof DivExpr) ||
   (be instanceof RemExpr))
  { ... }
```

Because OOMatch currently lacks the ability to do disjunctive pattern matching, these conditions could not easily be removed. If an "or" operator were introduced into patterns (a very worthwhile addition that is briefly discussed later in the future work, Section 9.3.2), these cases could also conveniently be factored into separate methods.

There were also several empty methods in the original Java version that were made unnecessary by the automatic dispatch of OOMatch. A disadvantage of the OOMatch version is that the original Java version contains error reporting for unhandled AST nodes, to help find bugs in ConstraintChecker. This error reporting is easy to do in the Java version because it can simply appear in the "else" case of the if-else block checking the AST node type. In the OOMatch version, doing such error reporting would involve adding many extra cases, because some of the AST node types only match for particular patterns. For example, an `ArrayRef` might be passed as a parameter, but it might not be the particular form of `ArrayRef` that requires processing. In the Java version, the "else" clause with the error message would not be executed in this case. In the OOMatch version, if the most general method were to contain an error message, it would be incorrectly executed in the case where the parameter is an `ArrayRef` but not of the particular form requiring processing. This error reporting was therefore omitted.

Another issue that arose is that as cases were split into separate methods, these methods sometimes need to access variables that were defined within the original method, but not in the block of code being factored out. For example:

```
    //Java
    public void caseAssignStmt(AssignStmt stmt)
    {
        Value l = stmt.getLeftOp();
        Value r = stmt.getRightOp();
        TypeNode left = null;
        TypeNode right = null;

        //******** LEFT ********

        if(l instanceof ArrayRef)
        {
            //Use stmt
        }
    }
```

This requires `stmt` to become an extra argument to the new sub-method handling `l`. Ideally, in a language with nested methods, these new methods would become nested inside the original method, and these extra parameters would not be needed.

Despite these minor issues, OOMatch was found to greatly simplify a significant amount of this complex AST traversal within Soot. In addition to a reduction in code size, there was a significant reduction in the amount of casting and `instanceof` uses in the class, the `instanceof`s dropping from 121 to only 35. In addition, the OOMatch version more naturally expresses the rule-based nature of the code, and allows the ability to extend (rather than modify in-place) particular methods or cases.

Chapter 8

Other Related Work

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

8.1 Scala

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

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

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

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

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

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

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

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

8.2 OCaml

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

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

8.3 TOM

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

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

8.4 Views

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

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

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

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

8.5 JMatch

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

For example:

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

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

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

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

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

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

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

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

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

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

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

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

8.6 JPred

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

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

8.7 Other Approaches to Pattern Matching as Dispatch

HydroJ [LLC03] introduces a form of dispatch similar to ours, 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 compiles and runs, and silently ignores 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 does not provide a means of decomposing a Java type and safely extracting its internals to perform the match.

Extensible ML [MBC04] also allows dispatch on nested subpatterns. The goal of the language, however, is not dispatch, but an attempt to allow classes to be extended (which OO languages allow while functional ones generally do not), while simultaneously allowing functions to be extended with new cases (which functional languages allow but OO languages generally do not). It does not contain any form of deconstructor or way of decomposing an object safely, but contains a record-like dispatch in which an object must be deconstructed into its fields.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

Pattern matching as dispatch has been found to naturally subsume standard polymorphic dispatch and multimethods. Where those approaches only allow methods with subtype parameters to override those with supertype parameters, pattern matching as dispatch allows more specific *patterns* to override less specific ones. While occasionally useful in everyday programming, this becomes very useful in situations with large class hierarchies, where some part of the program is more conveniently and directly expressed as a set of rules rather than step-by-step instructions. Unlike traditional pattern matching done in "case" statements, our matching allows the ordering of cases to be determined automatically by the compiler. It also allows extension and adaptability by subclasses, which is impossible with a "case".

To demonstrate pattern matching as dispatch with an implementation, an object-oriented language, rather than a functional language with pattern matching, was chosen as a base, because matching becomes interesting and challenging when done on user-defined classes. This challenge is due to the need to allow the conflicting goals of both extracting the internals of an object and retaining encapsulation of the implementation details of the class, solved by our deconstructors. Java specifically was chosen as a base because of the availability of Polyglot, which allows features to be added without writing a compiler from scratch, as well as its support for subclasses and polymorphic dispatch.

The design of OOMatch has revealed many issues and nuances that pattern matching as dispatch raises. The difficulty of new errors introduced leads to a tension between safety and flexibility. We have found a balance between these goals, with a bias towards flexibility rather than safety. Rather than finding all errors during compilation, we have chosen a design that allows most programs that make sense, and can find ambiguity errors in all but three specific situations (Section 3.5). Two of these are very easy to avoid; ambiguities caused by multiple inheritance (Section 3.5.1) pose the greatest problem, not only to OOMatch, but to any language with both multimethods and multiple inheritance. Most of these errors, however, could be found with a whole-program analysis, perhaps even an analysis that warns the programmer of situations where such errors *might* be present.

Another tradeoff addressed was backwards-compatibility with Java code; we chose to give OOMatch a simple syntax at the cost of full backwards compatibility (Section 3.14). However, practically all the functionality of Java is supported, with identical syntax. Migration or use of Java code is simple, particularly due to the fact that differences between the Java and OOMatch versions can be found by the OOMatch compiler, as either warnings or errors.

While OOMatch has been shown, in Chapter 7, to be of some independent use, ultimately a programming language should be designed as a cohesive whole. In a language designed from the start to support both the declarative nature of functional programming and the encapsulation and polymorphism of object-oriented programming, pattern matching as dispatch could become significantly more natural and useful. In particular, fewer errors would need be left until run-time, as the intended deterministic nature of deconstructors could be enforced statically. The specification of how to decompose objects could also likely be made much simpler with a more declarative style, and with the support of such features as built-in tuples. As well, if implemented as a natural part of a language rather than as a translation to Java, many of the limitations we have discussed throughout this thesis could be trivially overcome.

Overall, OOMatch provides a balance of power lying between multimethods and general predicate dispatch. In doing so, it uses a simple, intuitive syntax that allows the easy expression of high-level ideas for rule-based systems, and it yields greater abstraction and extensibility.

9.2 Design Goals

In this section we note places in the thesis that show how our design goals, listed in Chapter 1, were achieved, or the degree to which they were achieved.

Flexibility was generally achieved by the addition of a considerably more powerful means of doing dispatch in Java. Since pattern matching as dispatch subsumes Java's dispatch, OOMatch is at least as powerful as Java. The only concern is backwards compatibility, which was mostly maintained; the differences described in Section 3.14 are minor and can be easily worked around. Certainly OOMatch provides all the features of Java, with all Java's options for programming styles, and as it introduces a whole new programming style that programmers can now use (pattern matching on objects and multiple method cases for different object patterns), the programmer's toolbox has expanded. Specific examples of flexibility in OOMatch are deconstructors (Section 3.2), which can contain arbitrary code and a boolean return value to restrict matching arbitrarily, static deconstructors (Section 3.13), which allow deconstruction of existing libraries and code bases (used on Soot in Section 7.2), and a simple way of manually specifying override relationships when the automatic ones chosen by the language do not suffice (Section 3.11).

The simplicity of our formulation of pattern matching as dispatch is evident in the subtyping semantics, and parameters with subpatterns being considered more specific than plain parameters, which fits naturally with Java's heavy emphasis on subclasses and polymorphism. Subcomponents of an object are often clearer and easier to write than equivalent patterns written in predicate dispatch clauses (Sections 2.2.2 and 8.6). The access specifiers in constructors (Section 3.1) make it very easy to allow matching of objects, particularly for programmers first learning the language. Specific examples of how OOMatch can simplify code are shown in the improved quality of the code of a Soot class (Section 7.2), the potential simplification of the Polyglot framework (Section 7.1), as well as other examples shown in Appendix B.

The safety of the language was achieved in the compiler checking to prevent ambiguity and no-such-method errors. Most of the ambiguity errors are found by the compiler, even if no calls are made to the ambiguous methods. In order to have a more flexible language, some ambiguity errors had to be deferred as run-time checks (Section 3.5); however, these specific cases can be avoided fairly easily, and extra care can be exercised when those situations do arise. No-such-method errors are all prevented by the compiler, unless `inc` methods (Section 3.1) are used; this is OK because `inc` is specifically a directive to circumvent the safety check.

The modular typechecking of Java was retained, and is handled by Polyglot.

The goal that programmers should only pay for what they use was not fully achieved, because all methods need a dispatcher associated with them due to implementation difficulties (Section 5.5.2); hence, all classes have overhead, even if they don't use pattern matching. However, as mentioned, this overhead could potentially be removed by a good optimizer; further, there is nothing in pattern matching as dispatch itself that forces overhead in all classes. The prototype compiler in general probably results in code that is quite slow; however, performance in general was not one of the goals of the implementation.

9.3 Future Work

Some possible future directions that pattern matching as dispatch could take are as follows. Not all of these are feasible to add to the current implementation - some are future directions for pattern matching as dispatch in general, rather than future directions for OOMatch.

9.3.1 Other data types

We have done pattern matching on classes because they are the most useful data type, especially in Java. But there are many other built-in or user-defined types in other languages. If patterns were extended to include those types as well, dispatch could potentially become much richer.

Obvious examples are tuples and variant types (or Scala's case classes [OAC$^+$06]), as pattern matching is already done on these in functional languages like ML. Variants (or case classes) are particularly interesting because they would make it much more worthwhile to drop the requirement that there must be a most general Java method, and instead do exhaustiveness checking to ensure that all cases of the variant are handled.

Subrange types would be another user-defined type that pattern matching as dispatch would benefit greatly from. It would be quite convenient to have a method that takes an integer in the range 1 to 10 which overrides a method that takes an integer in the range 1 to 20.

Perhaps even more interesting, and challenging, would be functions which themselves take patterns containing functions as parameters. The question of which function patterns are most specific could potentially be challenging to answer. Perhaps partial patterns could be allowed: for example, one pattern could accept a function with any parameters, while another could accept any function with at least one "int" parameter, and the latter could override the former.

Finally, arrays, or, more generally, containers, would likely provide the most useful patterns. Being able to specify elements of containers (as TOM [MRV03] allows) in patterns would not only allow ML-style list processing, but would allow precedence of these patterns to be determined automatically. Some possible syntax:

```
void f(int[]{0, ...}) { }
void f(int[]{int x, ...} arr) { }
void f(int[]{...}) { }
```

The first version of `f` would be the most specific, since it accepts only arrays of `int` starting with `0`. The second version is still more specific than the third, because it only matches arrays with at least one element, while the third version also matches arrays of size 0.

Additionally, the use of `...` could allow methods with a variable number of parameters to be specified using an array.

Container matching would require the language having some notion of what constitutes a container (so that user-defined containers could be matched as well), which is not very clearly specified in Java (as java.util.Collection is treated differently than the built-in arrays).

9.3.2 Disjunctive Patterns and Regular Expressions

Regular expressions are generally thought to be the most powerful form of pattern matching available, so it is natural to ask which of their elements might work well in OOMatch. Since regular expressions are used to match strings, they do not map perfectly to object matching, but one feature in particular - an "or" operator - could be very useful in our pattern matching.

An "or" operator would allow a variable to have one of a few possible forms, any of which cause the same method to be called. For example:

```
void f(Circle s | Square s) { ... }
```

This would cause this version of `f` to be called if passed a `Circle` or `Square`. The type of `s` would be the most specific supertype of all the possible types (`Shape` in this case). Another possible use would be to extract a variable that always has the same type, but which can appear in multiple places within a pattern. For example, in an AST, an expression with a negative sign in front of it might be treated the same as one without:

```
void f(Neg(Expr e) | Expr e) { ... }
```

Specifying how overriding works in this context would require consideration.

9.3.3 Partial Deconstructing

It frequently happens that a class contains many components worthy of being deconstructed. However, often a pattern only needs a few of them. For example, in a compiler, an AST node for methods would contain many components. A programmer may be writing a pattern to match only methods named "main", and may want to ignore all the other components of the method object. In the current state of OOMatch, they are forced to give names to all the other components of the deconstructor, even though they are not using them in the method.

One solution to ease this problem would be to introduce _ as a possible subparameter (_ is not currently present in OOMatch). An _ would simply mean, "ignore this component, and match anything that the deconstructor returns in this position." To take the example of a (simplified) Method class:

```
class Method {
    deconstructor Method(Type retType, String name,
        List params, List throwTypes)
    { ... }
    void f(Method(_, "main", _, _)) { ... }
    ...
}
```

Another possible solution to the unwanted component problem would be *named parameters*:

```
void f(Method(name="main")) { ... }
```

Any parameter (regular parameter, pattern, or literal) could be prefixed with `var=`, where `var` is the name of one of the deconstructor's components. The above method would match any `Method` object whose deconstructor returns the value `"main"` from its `name` component, and any other values from the other components.

There are two notable advantages to named parameters over _ parameters. First, named parameters are likely clearer, because programmers do not often remember the position of a component within the deconstructor, but may well be reminded of what the component means if there is a name associated with it. Second, if components (other than `name`) are added to or removed from the deconstructor, the above pattern using named parameters would not need to be changed; the one using _ parameters would.

This feature would introduce extra design complexities. Perhaps the most obvious is that overriding relationships would need to be re-specified. If there is another version of `f` that deconstructs a `Method` object and only uses the `name` component, then the overriding would be obvious; but if the pattern refers to other components, overriding is not so obvious. So far we have assumed that patterns can only be preferred to each other if they have the same number of components; this would need revision and redesign.

Another issue arises when there are overloaded deconstructors. If there is another deconstructor named `Method` which returns a component named `name`, the compiler would not know which one to bind to the pattern.

Due to these complexities, the feature has been left for future work.

Appendix A

Example of typechecking algorithm

The DAG structure at each step through the algorithm is shown in Figures A.2 through A.6, for the code in Figure A.1.

```
class Point { ... }
class CoorPoint extends Point { ... }
public class RectTest
{
    static void f(null) {}
    static void f(Point(int x, int y) p) {}
    static void f(Point p) {}
    static void f(Point(0, 0)) {}
    static void f(CoorPoint(0, 0)) {}
}
```

Figure A.1: Typecheck algorithm example code

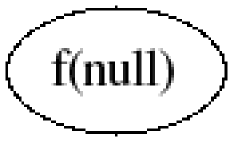

Figure A.2: Typecheck algorithm example, step 1

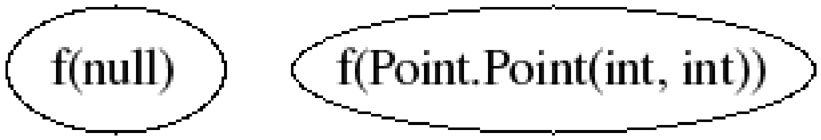

Figure A.3: Typecheck algorithm example, step 2

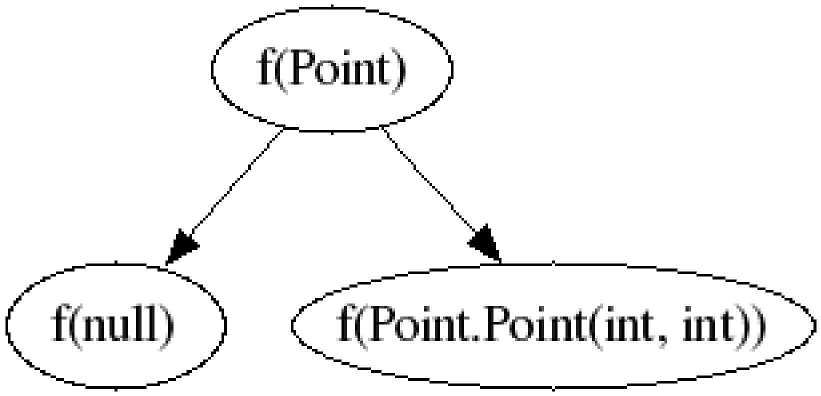

Figure A.4: Typecheck algorithm example, step 3

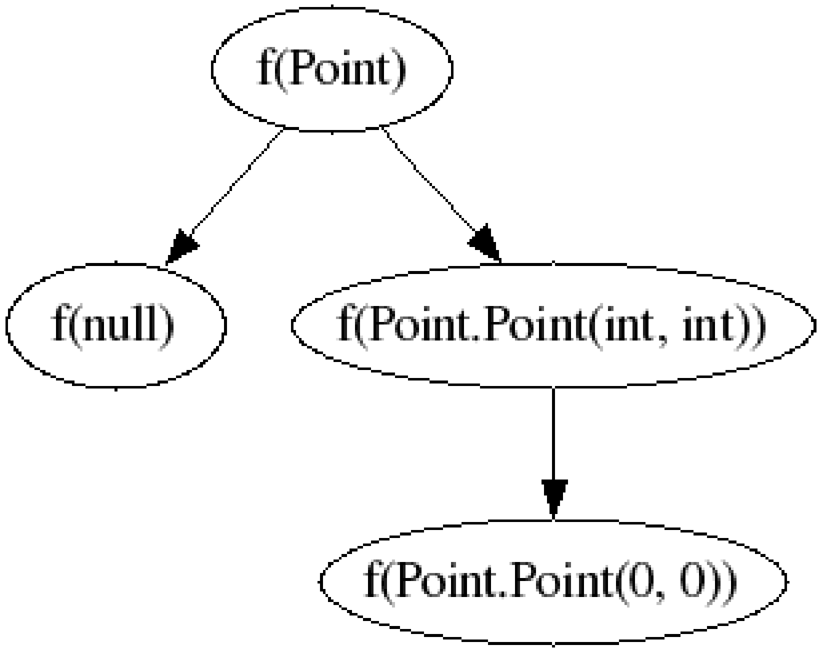

Figure A.5: Typecheck algorithm example, step 4

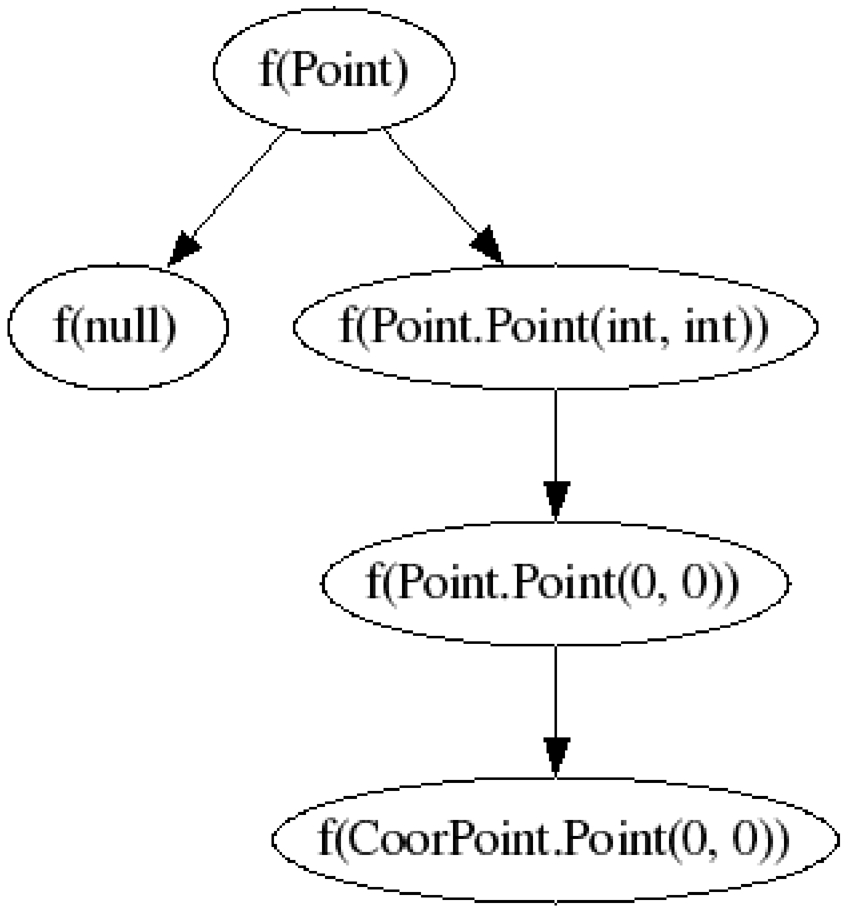

Figure A.6: Typecheck algorithm example, step 5

Appendix B

Examples of uses of OOMatch

Some examples of code found in the Polyglot compiler that could be simplified with the use of OOMatch are as follows.

- In the polyglot.ext.jl.ast.NodeFactory_c class, the following method can be found:

  ```
  public Assign Assign(Position pos, Expr left,
      Assign.Operator op, Expr right)
  {
      if (left instanceof Local) {
          return LocalAssign(pos, (Local)left, op, right);
      }
      else if (left instanceof Field) {
          return FieldAssign(pos, (Field)left, op, right);
      }
      else if (left instanceof ArrayAccess) {
          return ArrayAccessAssign(pos, (ArrayAccess)left, op, right);
      }
      return AmbAssign(pos, left, op, right);
  }
  ```

  This is a common case of where multimethods can remove run-time type tests. It could instead become:

  ```
  public Assign Assign(Position pos, Expr left,
  ```

```
    Assign.Operator op, Expr right)
{
    return AmbAssign(pos, left, op, right);
}
public Assign Assign(Position pos, Local left,
    Assign.Operator op, Expr right)
{
    return LocalAssign(pos, left, op, right);
}
public Assign Assign(Position pos, Field left,
    Assign.Operator op, Expr right)
{
    return FieldAssign(pos, left, op, right);
}
public Assign Assign(Position pos, ArrayAccess left,
    Assign.Operator op, Expr right)
{
    return ArrayAccessAssign(pos, left, op, right);
}
```

- In polyglot.types.ImportTable, there is the following method:

```
protected boolean isVisibleFrom(Named n, String pkgName) {
    boolean isVisible = false;
    boolean inSamePackage = this.pkg != null
            && this.pkg.fullName().equals(pkgName)
        || this.pkg == null
            && pkgName.equals("");
    if (n instanceof Type) {
        Type t = (Type) n;
        //FIXME: Assume non-class types are always visible.
        isVisible = !t.isClass()
            || t.toClass().flags().isPublic()
            || inSamePackage;
    } else {
        //FIXME: Assume non-types are always visible.
        isVisible = true;
    }
    return isVisible;
}
```

It could be simplified to this:

```
protected boolean isVisibleFrom(Named n, String pkgName) {
    //FIXME: Assume non-types are always visible.
    return true;
}
protected boolean isVisibleFrom(ClassType(Flags f), String pkgName) {
    boolean inSamePackage = this.pkg != null
            && this.pkg.fullName().equals(pkgName)
        || this.pkg == null
            && pkgName.equals("");
    return f.isPublic() || inSamePackage;
}
```

- In polyglot.ext.jl.types.PrimitiveType_c, there is this method:

```
public boolean numericConversionValidImpl(Object value) {
    if (value == null)
        return false;
    if (value instanceof Float || value instanceof Double)
        return false;

    long v;

    if (value instanceof Number) {
        v = ((Number) value).longValue();
    }
    else if (value instanceof Character) {
        v = ((Character) value).charValue();
    }
    else {
        return false;
    }

    if (isLong())
        return true;
    if (isInt())
        return Integer.MIN_VALUE <= v && v <= Integer.MAX_VALUE;
    if (isChar())
```

```
        return Character.MIN_VALUE <= v && v <= Character.MAX_VALUE;
    if (isShort())
        return Short.MIN_VALUE <= v && v <= Short.MAX_VALUE;
    if (isByte())
        return Byte.MIN_VALUE <= v && v <= Byte.MAX_VALUE;

    return false;
}
```

It could be simplified to:

```
public boolean numericConversionValidImpl(Object value) {
    return false;
}
public boolean numericConversionValidImpl(Number(long v)) {
    return inRange(v);
}
public boolean numericConversionValidImpl(Character(char v)) {
    return inRange(v);
}
public boolean inRange(long v) {
    if (isLong())
        return true;
    if (isInt())
        return Integer.MIN_VALUE <= v && v <= Integer.MAX_VALUE;
    if (isChar())
        return Character.MIN_VALUE <= v && v <= Character.MAX_VALUE;
    if (isShort())
        return Short.MIN_VALUE <= v && v <= Short.MAX_VALUE;
    if (isByte())
        return Byte.MIN_VALUE <= v && v <= Byte.MAX_VALUE;

    return false;
}
```

- polyglot.visit.ConstantFolder contains:

```
public Node leave(Node old, Node n, NodeVisitor v_) {
    if (! (n instanceof Expr)) {
```

```
    return n;
}

Expr e = (Expr) n;

if (! e.isConstant()) {
    return e;
}

// Don't fold String +.  Strings are often broken up for better
// formatting.
if (e instanceof Binary) {
    Binary b = (Binary) e;

    if (b.operator() == Binary.ADD &&
    b.left().constantValue() instanceof String &&
    b.right().constantValue() instanceof String) {

    return b;
    }
}

Object v = e.constantValue();
Position pos = e.position();

if (v == null) {
    return nf.NullLit(pos).type(ts.Null());
}
if (v instanceof String) {
    return nf.StringLit(pos,
            (String) v).type(ts.String());
}
if (v instanceof Boolean) {
    return nf.BooleanLit(pos,
             ((Boolean) v).booleanValue()).type(ts.Boolean());
}
if (v instanceof Double) {
    return nf.FloatLit(pos, FloatLit.DOUBLE,
               ((Double) v).doubleValue()).type(ts.Double());
    }
    if (v instanceof Float) {
        return nf.FloatLit(pos, FloatLit.FLOAT,
                    ((Float) v).floatValue()).type(ts.Float());
    }
    if (v instanceof Long) {
        return nf.IntLit(pos, IntLit.LONG,
                  ((Long) v).longValue()).type(ts.Long());
    }
    if (v instanceof Integer) {
        return nf.IntLit(pos, IntLit.INT,
                  ((Integer) v).intValue()).type(ts.Int());
    }
    if (v instanceof Character) {
        return nf.CharLit(pos,
                   ((Character) v).charValue()).type(ts.Char());
    }

    return e;
}
```

This could become:

```
public Node leave(Node old, Node n, NodeVisitor v_) {
    return fold(n);
}
public Node fold(Node n)
{
    return n;
}
public Node fold(Expr e)
{
    if (e.isConstant())
        return constFold(e.position(), e.constantValue())
    else
        return e;
}
//Expr.constantValue below is a deconstructor
public Node fold(Binary(Expr.constantValue(String l),
    Binary.ADD, Expr.constantValue(String r))) e)
{
    return e;
}

public Node constFold(Position pos, Object v)
{
    throw new InternalCompilerError("Unrecognized constant type: " + v);
}
public Node constFold(Position pos, null)
{
    return nf.NullLit(pos).type(ts.Null());
}
public Node constFold(Position pos, Boolean(boolean b))
{
    return nf.BooleanLit(pos, b).type(ts.Boolean());
}
...
```

Appendix C

Example dispatcher output

```
static int f(Point arg1, Class arg2) {
    int methodChosen = 0;  //variable to keep
        //track of which method was chosen so far.
        //Holds the ID of the chosen method.
    {  }
    {
        //Try to dispatch to f(null)
        {
            if (arg1 == null) {  //condition
                //required for applicability
                if (methodChosen != 0)  //if
                    //there was already a
                    //method chosen, it is an
                    //ambiguity
                    throw new Error("The following"
                     + "2 methods are ambiguous:\n" +
                     (messageFor1$(1) + (" and \n" +
                     messageFor1$(methodChosen))) +
                     " in class RectTest.\n");
                methodChosen = 1;  //otherwise, set
                    //the method chosen
            }
        }

        //Try to dispatch to f(CoorPoint(0, 0))
```

```
Object[] retVal$5$1 = null;  //declare
    //variable to hold deconstructor result

//Call the deconstructor, but only if the
//subtype condition holds.
if (arg1 instanceof CoorPoint &&

      (arg2 == null ||
       CoorPoint.class.isAssignableFrom(arg2) ||
       arg2.isAssignableFrom(CoorPoint.class)) &&
          //if either
            //of the static argument type and
            //potential match's parameter
            //type is a subtype of the other
      arg1 != null)  //prevent a crash
        //(can't call a deconstructor
        //on a null object)
    retVal$5$1 =
        ((CoorPoint) arg1).Point$2();
{
    //repeat the subtype check
    if (arg1 instanceof CoorPoint &&
          (arg2 == null ||
          CoorPoint.class.isAssignableFrom(arg2)
          ||
          arg2.isAssignableFrom(CoorPoint.class))
          && arg1 != null &&
          retVal$5$1 != null &&  //If the
            //deconstructor succeeded

          //and the subcomponents returned
          //match the pattern
          ((Integer) retVal$5$1[0]).intValue()
            == 0 &&  //Have to unbox
                //primitive types
          ((Integer) retVal$5$1[1]).
            intValue() == 0) {
        if (methodChosen != 0)
            throw new Error("The following"
            + "2 methods are ambiguous:\n" +
            (messageFor1$(5) + (" and \n" +
            messageFor1$(methodChosen))) +
            " in class RectTest.\n");
        methodChosen = 5;
    }
}

//Try to dispatch to f(Point(0, 0))
Object[] retVal$4$1 = null;
if (arg1 != null) retVal$4$1 = arg1.Point$2();
if (methodChosen != 5) {  //If the child
    //of f(Point(0, 0))
    //(namely, f(CoorPoint(0, 0)) hasn't
    //applied.  If it has, we shouldn't even
    //check this one.
    if (arg1 != null && retVal$4$1 != null
        && ((Integer) retVal$4$1[0])
            .intValue() == 0 &&
        ((Integer) retVal$4$1[1])
            .intValue() == 0)
    {
        if (methodChosen != 0)
            throw new Error("The following"
            + "2 methods are ambiguous:\n" +
            (messageFor1$(4) + (" and \n"
            + messageFor1$(methodChosen)))
            + " in class RectTest.\n");
        methodChosen = 4;
    }
}

//Try to dispatch to f(Point(int x, int y) p)
Object[] retVal$2$1 = null;
if (arg1 != null) retVal$2$1 = arg1.Point$2();
if (methodChosen != 5 && methodChosen != 4) {
    if (arg1 != null && retVal$2$1 != null) {
        if (methodChosen != 0)
            throw new Error("The following"
                + "2 methods are ambiguous:\n" +
                (messageFor1$(2) + (" and \n" +
                messageFor1$(methodChosen))) +
                " in class RectTest.\n");
            methodChosen = 2;
        }
    }

    //Try to dispatch to f(Point p)
    if (methodChosen != 5 &&
        methodChosen != 4 &&
        methodChosen != 1 &&
        methodChosen != 2) {
        if (true) {
            if (methodChosen != 0)
                throw new Error("The following"
                + "2 methods are ambiguous:\n" +
                (messageFor1$(3) + (" and \n" +
                messageFor1$(methodChosen))) +
                " in class RectTest.\n");
            methodChosen = 3;
        }
    }

    //Call the appropriate method,
    //passing only the values of
    //the names declared in patterns.
    switch (methodChosen) {
        case 1: return f$1();
        case 5: return f$5();
        case 4: return f$4();
        case 2:
            return f$2(arg1,
            ((Integer) retVal$2$1[0]).intValue(),
            ((Integer) retVal$2$1[1]).intValue());
        case 3: return f$3(arg1);
    }
}
```

```
    //If no method matched, throw an error.
    //This statement is usually unreachable.
    throw new Error("No method found for call.");
}
```

Bibliography

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

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

[CMC$^+$06] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

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

[CUP96] CUP LALR parser generator for Java, 1996. Available at http://www2.cs.tum.edu/projects/cup/ on 23 May 2007.

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

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

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatch: A Unified Theory of Dispatch. In *ECOOP '98, the 12th Eu-*

ropean Conference on Object-Oriented Programming, pages 186–211, 1998.

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

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

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

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

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

[JED] JEdit Programmer's Text Editor. Available at http://www.jedit.org/ on 26 June 2007.

[Joh79] Steven C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[LLC03] 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.

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

[MBC04] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

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

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

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

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM Press.

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

[OZ05] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

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

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

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Indianapolis, IN, 1997.

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

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

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