Adding functional style pattern matching features to object-oriented languages

Joseph Thelen
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
thele116@morris.umn.edu

ABSTRACT

Pattern matching as a major feature in programming languages is particularly interesting because is it has long been a mainstay feature of prominent languages such as Scala, Haskell, and Erlang. However, its use has been limited primarily to regular expressions for use on strings in other popular languages such as Java, C++, and Perl. Pattern matching has cited as one of the major reasons for choosing a functional language, such as Haskell or Erlang. As such, efforts have been made to bring pattern matching to languages that did not previously incorporate it. Here, we will discuss two such attempts, one in C++ and one in Java, briefly comparing them.

Keywords

Pattern matching, C++, Java, Dispatch, Haskell, Types, Objects, Multimethods, Multiple Dispatch, Regex, Regular Expression

1. INTRODUCTION

1.1 What is pattern matching?

Pattern Matching, at least as intuitively understood, is a relativity simple concept. A dictionary provides us with a definition of a "pattern" as a "repeated form or design" or "an original of model considered for or deserving of imitation", and of "matching" as "corresponding or causing to correspond in some essential aspect" or as "equal in number or equivalent." Armed with these definitions, we can easily recall examples of common place pattern matching. For instance, the game that children play in which various geometrically shaped blocks must be placed into corresponding holes in a box could be considered pattern matching.

1.2 In Programming

Looking at pattern matching in the context of computer science and programming we can start by simply considering the opening line of the Wikipedia article on the subject, which defines pattern matching as "the act of checking a given sequence of tokens for the presence of the constituents of some pattern."[10] Putting this in terms of our block-game example, blocks, perhaps strung together in a sequence, could be considered *tokens*, and a *pattern* would be a sequence of blocks that we are looking for, represented by the sequence of holes in the box. In programming, we will need some syntax for representing the patterns we are looking for.

Regular expressions, which are quite commonplace in programming and computer science, are one example of pattern matching in programming in that they are often used to define patterns against which strings can be matched. More interesting than string parsing though, at least to us, are usages of pattern matching as a major feature of programming languages as opposed to as stand alone or domain-specific tools. In some programming languages / implementations pattern matching can be used to handily decompose data structures, patterns can be used in defining functions, and sometimes patterns are even made first-class-citizens, such as in the `first-class-patterns` package for Haskell[1]. *First-class-citizen* here means that patterns are treated in the same way that we might treat Strings or integers in Java; they can be passed around as arguments, assigned to variables, and returned by functions.

Overall, pattern matching is a desirable language feature because it adds a great deal of flexibility in how programmers can think about and compose functions and deal with data. Pattern matching is often an integral part of technology such as natural language processing, and plays a crucial role in languages that rely on message passing or channels, such as Go or Erlang/Elixir. Pattern matching is cited as one of the top reasons that programmers would choose to use a functional language [8].

1.3 Simple Examples in Haskell

Having covered some basic definitions, let us consider some simple examples to get a better idea of how pattern matching can appear in programming. Our examples here are in Haskell. Haskell is a functional programming language which is statically typed, features pattern matching quite heavily, and has a relatively easy-to-understand syntax. Here, we provide a definition for a function `ourFunction`, which should take in a number and return the result of adding one to that number:

```
ourFunction x = x + 1
```

[1]available: hackage.haskell.org/package/first-class-patterns

.

In this situation the "=" sign is used to signify not an assignment, but a definition. The function name is provided, followed by each of the function's parameters separated by spaces (in this case a single parameter, `x`). The body of the function follows the "=" sign. Currently, `ourFunction` will return `x + 1` whenever it is passed a single number, `x`. It is important to note here that, although Haskell is statically typed, we are not required to specify the types of our function - they will be inferred during compilation time. `ourFunction` here is limited to taking in and returning numbers.

The power of pattern matching comes into play when we want our function to consume more complicated information, or when we want to add more complex behavior. For example, suppose we want to add a special case for an argument of `0` to our function. Such a modification might look like this:

```
ourFunction 0 = -1
ourFunction x = x + 1
```

We've added a new definition for the function `ourFunction` in which an argument of `0` will return `-1`. In Haskell, all definitions for a function will be used, so we are not 'overwriting' our existing definition by adding this new one. When `ourFunction` is called, Haskell will attempt to match the provided arguments against each definition it has for `ourFunction`, in the order they were defined, until it finds a definition that will accept the arguments (which is then used). This is an example of pattern matching affecting dispatch[2], where *dispatch* refers to deciding which definition to use when calling a function. Of course, this function and its behavior could easily be reproduced in another language using switch statements or if-conditionals, however doing this becomes increasingly difficult as the complexity of patterns grows. Of note is the fact that we would *need* to use something like an if-conditional if we were adding this behavior to an equivalent function in Java. While Java does allow overloading of functions ('methods' in Java), this is only allowed when those function definitions differ in number of arguments or argument type(s)[5]. Similar rules apply to overloading in C++. Here, both of our definitions for `ourFunction` take a single argument of the same type.

Suppose that we want to rewrite `ourFunction` to accept a *list* of numbers and return the result of multiplying the first element of the list by the sum of the rest of the list, returning `-1` in the event of an empty list. The code might look like this:

```
newFunction [] = -1
newFunction (x:xs) = x * sum xs
```

In our first definition for `newFunction`, we match an empty list and return `-1`. The ":" symbol used in the second definition indicates a "cons", or the adding of an element to the beginning of a list. For example, `5:[4, 3, 2]` would result in `[5, 4, 3, 2]`. In the second definition, using the aforementioned "cons" operator, we match any list which can be broken into a single element and another list - returning the result of multiplying that list's first element by the sum of the rest of its elements (using a call to the built-in `sum` function with the list `xs` as an argument).

As previously mentioned, pattern matching can be used to decompose data in useful ways. This implementation of `newFunction` provides us with an example of how we might decompose a list using pattern matching. In the second definition of `newFunction` the pattern `(x:xs)` will match any list which can be broken down into a single element and a list, the variables `x` and `xs` being then bound to the first element of the list and the rest of the list, respectively, in the body of that function definition. For example, calling `newFunction` on the list `[4,3,2,1]` matches the second definition's pattern, assigning `x` the number 4 and `xs` the list `[3,2,1]`, ultimately returning the number 24.

In an effort to further demonstrate the capabilities of pattern matching, let us add one additional special case where any list with the number `0` as its first element will skip the multiplication step and instead return only the sum of all elements in the list, as well as implementing our own summation rather than using the built-in `sum`:

```
newFunction [] = -1
newFunction (0:xs) = ourSum xs
newFunction (x:xs) = x * ourSum xs

ourSum [] = 0
ourSum (x:xs) = x + ourSum xs
```

Remember that, as previously mentioned, pattern matching will bind matched parts of the data being matched against to the associated variables in the pattern.

We've also added a new function, `ourSum`, which replaces our use of the built in `sum`. The function `ourSum` is recursive, meaning that it makes calls to itself. In explanation, consider a call to `ourSum` with the list `[1,2,3]` as the argument. In this case, we would skip over the first definition of `ourSum` because our argument is not an empty list, instead matching the pattern of the second definition and breaking our argument into the number `1` (`x`) and the list `[2,3]` (`xs`). This call to the second definition of `ourSum` will return `1 + ourSum [2,3]`, the sum of the number 1 and the result of another call to `ourSum` with the list `[2, 3]` as its argument. This second call will again match the second definition, breaking the list `[2, 3]` into the number `2` and the list `[3]` and returning `2 + ourSum [3]`. The third call here, as before, match the second definition's pattern and break its argument out into the number `3` and an empty list, `[]` - returning `3 + ourSum []` which will become `3 + 0`. In the end, our original call will return `1 + 2 + 3 + 0`, which evaluates to 6.

Finally, note that calling `newFunction` on any list of any single number will return that number multiplied by the sum of an empty list (which is zero). To avoid performing any unnecessary function calls in these cases, we can add one final definition for `newFunction` which matches lists with only one number in them and returns `0`:

```
newFunction [] = -1
newFunction [_] = 0
newFunction (0:xs) = ourSum xs
newFunction (x:xs) = x * ourSum xs
```

Our new definition, now the second one, makes use of a *wildcard* - denoted with '`_`'. A wildcard will match any single thing, but will not perform any binding. This is perfect for our new definition because we only care that our argument is a list with a single element, not about what that element is.

[2]In Haskell's terminology, this is called "Dispatch Control".

2. DEFINITIONS AND EXAMPLES

Before we can discuss pattern matching implementations, it is important to have at least a casual familiarity with some of concepts associated with pattern matching.

2.1 Multi-methods and Multiple-dispatch

Multiple-dispatch and multi-methods are pieces of terminology relating to language features commonly associated with pattern matching, or at least languages which feature pattern matching. Dispatch, in this context, refers to the process of deciding which implementation of a given method-/function to actually use when called. Take our previous Haskell examples, for instance: we define the same function multiple times, and the definition which gets used depends on what arguments we call it with. The term *multiple dispatch* refers to the ability to actually use multiple definitions for a single function, and the term *multi-methods* refers to methods (or functions) which take advantage of this capability by having multiple definitions. An important distinction to make is that while the Java method overloading discussed in section 1.3 takes place during compile time, dispatch takes place duing runtime.[1][9][4]

2.2 Algebraic Types

One major obstacle faced by those wishing to add functional pattern matching features to languages like C++ and Java is that those languages do not naively support algebraic types, which are useful in pattern matching. Algebraic types are, essentially, just new types created by combining existing types using "algebraic" operations such as plus, minus, logical and, or, multiplication, etc. [2]. This is useful in functional languages like Haskell, where a large proportion of data will be represented as lists and tuples. This functionality is desirable because it allows the programmer to more freely represent how data can be organized, decomposed, and matched against. For example, in Haskell a singly-linked-list, where the list can be either empty, or composed of a single element added to another list, might be defined as follows:

```
data LList a = Empty | Cons a (LList a)
```

Here, "`data`" indicates that we are defining a type, `LList` is the name of our type, and `a` is a *type parameter* which is used to determine the type of a given instance of the `LList` type. The "`|`" symbol is an "or", meaning that the type `LList` has two *variants*: `Empty` and `Cons`. Each variant has a *constructor*, for our purposes the code following its name, which specifies what the constructor will accept. For example, the `Cons` constructor in our example wants a single thing of type `a`, and something of the type `LList a`. This kind of type lends itself to pattern matching and can be easily decomposed. For example, looking at our earlier Haskell examples, specifically the second definition of the `sum_helper` function, we find a list being matched against the pattern `(x:xs)`. This pattern matches a list that can be decomposed into a single element plus another list, which is essentially what the second variant of our `LList` is.

3. ADDING PATTERNS TO C++

While some previous efforts have been made to add pattern matching to C++ in various forms, such as Prop: "a C++ based pattern matching language" [3], one of the more comprehensive efforts of late is the *Mach7* library by Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup in 2013, summarized in their paper *Open Pattern Matching For C++* [8]. Their effort is particularly interesting because they strove to add "functional-style" pattern matching which is type safe, makes patterns first-class citizens, allows user-created patterns, and is implemented as a library. This approach, and the scope of features planned, is somewhat unique compared to other such projects some of which, like *"Prop"*, tend to be focused on compiler development and shy away from a library implementation [3].

3.1 Example

Many of the code examples presented by Solodkyy et al. are rather intimidating at first glance, especially to those not familiar with C++, due to some expectation of familiarity with both C++ and the pattern matching as a concept. That said, there is a small selection of simple examples which let us start to compare code written using this library to our previous Haskell examples. One such block of code, a recursive definition of factorial, is quite easily compared to our Haskell examples and is reproduced here:

```
int factorial(int n) {
 unsigned short m;
 Match(n) {
  Case(0) return 1;
  Case(m) return m*factorial(m-1);
  Case(_) throw std::invalid_argument("factorial");
 } EndMatch
}
```

In this example, the "_" symbol serves the same purpose as in our earlier Haskell example: as a *wildcard*. Although the code here only defines `factorial` once, as opposed to the multiple function definitions we see in our Haskell example, the similarities between the code contained with the *Match* block here and our multiple function definitions is quite apparent. Note that the local variable `m`, which is used in the pattern of the second `Case`, is an unsigned short while the data we will be matching against is an integer. The second `Case` will only match if the integer we are matching against can be converted into an unsigned short, otherwise the functions throws an error.

3.2 Features and Implementation

3.2.1 Algebraic Data Types

C++ does not have built in support for algebraic types. The authors of *Mach7* were able to encode algebraic types in C++ by using abstract classes to represent the algebraic data type (in our Haskell example this would be the type `LList`) and derived classes to represent the variants. However, constructors in C++ do not allow themselves to be easily re-purposed for pattern matching. To get around this, the authors have added some more explicit syntax for using them in pattern matching. This extra syntax is visible in the the `Case` lines of following example, adapted from an example provided for *Mach7* to resemble our Haskell example, of a function to check the equality of two linked lists [3].

```
struct LList          { virtual ~LList(){} };
```

[3]This example is based on the lambda term equality example in Solodkyy et al. and has not been tested or confirmed to work.

```
struct Item : LList { std::string item_name};
struct Empty: LList { NULL };
struct Cons : LList { Item& item; LList& rest};

bool operator==(const LList& left, const LList& right) {
  var⟨const std::string &⟩ s; var⟨const LList&⟩ x;
  var⟨const Item&⟩ y;
  Match(left, right) {
    Case(C⟨Item⟩(s),      C⟨Item⟩(+s))      return true;
    Case(C⟨Empty⟩(NULL), C⟨Empty⟩(NULL)) return true;
    Case(C⟨Cons⟩(y, x),  C⟨Cons⟩(+y,+x)) return true;
    Otherwise()                            return false;
  } EndMatch
}
```

The additional syntax needed to use constructors in pattern matching is described by the authors as being of "the form $C\langle T_i\rangle(P_1...P_{m_i})$, where T_i is the name of the user-defined type we are decomposing and $P_1...P_{m_i}$ are patterns that will be matched against members of T_i." Also of note is the use of the "`+`" symbol in the third case of our *Match* block. Here, '`+`' is what the authors refer to as an *equivalence combinator*: Note that the variables `y` and `x` have already appeared once in our pattern before the occurrence of `+y` and `+x`. The pattern will only match if the values matched by the second (with `+`) occurrences of `y` and `x` are equal to the existing values of those variables, which in this case have already been bound to `y` and `x` earlier in the pattern. Note there is a chance that these checks for equality will result in recursive calls to the overloaded `==` function when checking equivalence of the two occurrences of `x` in case 3. There are several other pieces of code that would need to be written before this example could be run, namely the *bindings*. Avoiding too much technical detail, *Mach7* requires the programmer to define *bindings* for each class hierarchy they create so that those classes can be properly decomposed during matching. This is necessary because classes in C++ can have multiple constructors and thus we cannot presume to use a classes' constructor as its deconstructor. In *Mach7* this is accomplished by "specializing the library template class bindings"[8] where *bindings* is a *template class* provided by the *Mach7* library. Defining a template class allows the programmer to specify the behavior of a class without concerning themselves with what types the class with handle; similar to the way Java Generics are used.

3.2.2 Patterns

The approach the authors of *Mach7* take is not to simply implement patterns as objects. Rather, they implement patterns as *expression templates* which are "composed at compile time" rather than during run time as would be the case for objects, allowing the types of patterns to be checked at compile time and offering some performance benefits. In short, *expression templates* are a feature of C++ which allow us to introduce some laziness into the evaluation of expressions during run-time by doing some evaluation at compile time. Laziness means that we avoid fully evaluating an expression - instead only evaluating it as much as is needed at the time. For instance, if we are matching against a pattern which breaks an arithmetic expression with a binary operator (such as `(1 + (2 + 5))`) down into two sub-expressions, we need not fully evaluate the sub-expression `(2 + 5)` during pattern matching.

Important to note is the fact that patterns in *Mach7* must conform to two constraints: `PATTERN` and `LAZYEXPRESSION`. The `PATTERN` constraint requires an expression to be copyable and a predicate on its subject type - meaning that a pattern must include a means by which to check if it is applicable to a given subject. A `LAZYEXPRESSION` must be copyable, and must provide a function to evaluate the result of the expression.

3.2.3 Matching

As discussed in our Haskell examples, in many cases the things we do with pattern matching *appear* very similar to things we might typically do with switch statements. In *Mach7*, their `Match` functionality is actually implemented as an extension of the "efficient type switch for C++", which happens to fall under the *Mach7* project's umbrella and is described in another paper by Solodkyy et al.[7]. Briefly, the efficient type switch is an implementation of the switch statement which allows us to switch based on the run-time types of objects. The efficient type switch makes use of hashing and caching to increase its performance, and its speed is supposedly one of the major advantages of pattern matching in *Mach7*.

3.3 Results

3.3.1 Performance

The authors conclude with an evaluation of the performance of their solution, comparing both their chosen solution (patterns as expression templates), and an alternative "patterns as objects" approach, to functionally equivalent hand-optimized code which lacks any pattern matching. Significant measures are taken to ensure that their patterns-as-objects implementation does not suffer typical performance pitfalls such as allocating objects on the heap. Comparisons of the median run times of several algorithms showed that an implementation with patterns as *expression templates* (*Mach7*) has substantially less overhead (compared to the functionally-equivilant baseline) than a patterns-as-objects implementation. There were no cases in which the patterns-as-objects implementation had less overhead than the patterns as expression templates implementation, with the latter frequently having overhead more than an order of magnitude less than the former. For example, there was a 395% overhead for patterns-as-objects verses a 15% overhead for patterns as expression templates on a function calculating fibonacci numbers.

Compile times were also evaluated in the same manner, and it was found that compilation times are only lightly impacted by either implementation, with both pattern matching implementations actually performing up to 10% better in a small handful of cases. To quote Solodkyy et al.: "the difference in compilation times was small: on average, 3.99% slower for open patterns and 4.84% slower for *patterns as objects*."

3.3.2 Usability

The usability of the features added by the *Mach7* project was tested by means of helping a programmer re-write some Haskell code in C++ using *Mach7*. Overall, the exercise was successful. The authors note that *Mach7* has an advantage over Haskell when defining certain kinds of patterns because "Haskell does not support equivalence patterns or an equivalence combinator and had to use guards to relate different arguments" [8]. It's not all good news though, as *Mach7* becomes more complicated than Haskell in the body

of the function after pattern matching. This is due primarilly to the fact that C++ requires programmers to explicitly manage memory. The authors also note that the patterns in *Mach7* are not *actually* first-class-citizens because "one cannot create a run-time data structure of patterns (e.g. a composition of patterns based on user input)" [8].

4. ADDING PATTERNS TO JAVA

The motivation to add functional style pattern matching features to Java is primarily the same as in C++: bringing a desirable feature found in functional languages to a popular language which lacks it. Interestingly enough, though, in at least one prominent project the reasoning seems to be that we should add useful features of functional languages to Java because, while they have their benefits, those functional languages lack crucial object oriented features found in Java. Said project is "OOMatch", an effort by researchers at the University of Waterloo to add pattern matching as dispatch in Java [6].

4.1 Example

In the previously mentioned paper describing the OOMatch project, an example is given which implements a simple optimizer for arithmetic expressions. The code from that example is shown in Figure 1. It consists of a handful of classes representing some possible components of arithmetic expressions, followed by some examples of methods which make use of those classes and features of OOMatch.

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

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

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

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


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

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

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

Figure 1: OOMatch Example

Here, we can see many similarities to our earlier Haskell examples. The method `optimize` is defined multiple times, and in a way which would not be feasible using only Java's existing mechanisms for method overloading: all of these definitions take a single argument, which is assumed to always be of type `Expr`.

For clarity, consider that an example of an expression constructed based on this class hierarchy might look something like `(1 + (2 + 5))`. This expression (`Expr`) can be referred to more specifically as a `Binop`, and more specifically yet as an instance of `Plus`. This instance of `Plus` contains a `NumConst` (the numeric constant 1) and another instance of `Plus`. Should we pass our example expression to `optimize`, we will match only the first definition's pattern, as our expression cannot be decomposed into a `Plus` containing an `Expr` and the numeric constant 0, and cannot be decomposed into a `Binop` containing two numeric constants. However, if we were to pass in only the sub-expression `(2 + 5)` we would match not only the first definition's pattern, but also the third's. Here it is prudent to note that in the OOMatch project, in contrast to Haskell, matching is not done in the order that definitions for a function appear. Instead, in OOMatch, matching is done in order of how specific each pattern is. So in the case of the expression `(2 + 5)`, we would end up falling into the third definition of `optimize` because its pattern is the most specific.

4.2 Features and Implementation

Java being, as some might say, a heavily object oriented language, it is only natural that an attempt to implement pattern matching in Java would focus on objects. Indeed, this is the case with *OOMatch*.

4.2.1 Algebraic Data Types and Decomposition

The OOMatch project does not directly make an attempt to encode algebraic data types, as the *Mach7* project did, because "simple algebraic types and tuples aren't used much in object-oriented programming" [6]. The OOMatch project goes about adding pattern matching to Java by modifying grammars from the Java language specification in order to "[add] deconstructors as a new kind of class member." These *deconstructors* are a major part of OOMatch. We've discussed previously how pattern matching can be used to decompose (or deconstruct) data, specifically in our Haskell examples. Deconstructors are also mentioned in our discussion of the *Mach7* project (§3.2.1). Here, deconstructors are added to classes in order to break down an object structurally and expose certain values for pattern matching.

OOMatch provides programmers with two ways of adding deconstructors: The first method simply requires the programmer to "add access specifiers to constructor parameters", which is as simple as it sounds: the programmer needs to add an access speifier such as '`Public`' in front of each of the prarmeters in their constructor, as seen in the following example of a definition for the `Binop` class, used in Figure 1, taken from Richard et al.:

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

Alternativley, deconstructors can be added to a class by adding a specific deconstructor method to the class, which "breaks down `this` into components and returns them to be matched against" [6]. A deconstructor for the `Binop` class introduced in Figure 1 might look like this:[4]

```
deconstructor Binop(Expr e1, Expr e2) {
        e1 = this.e1;
        e2 = this.e2;
}
```

[4]Taken directly from an example given in [6]

4.2.2 Patterns and Matching

In OOMatch a class specifies, via its deconstructors, how it can be decomposed. A pattern used in a method definition is simply an attempt to decompose its arguments in a specific way, and either succeeds or fails.

4.3 Results

4.3.1 Performance

The authors of the OOMatch project provide many proofs of the correctness of their implementation, but do not provide any kind of practical performance analysis. Presumably, since code written using the OOMatch project compiles to standard JVM bytecode, the project's effect on performance is not large. However, the *Mach7* project did note the inferior performance of solutions such as that used in OOMatch, which does most of its work at run-time. Additionally, the OOMatch project notes that there are a number of potential pitfalls, all resulting in thrown exceptions, which a programmer might face if introducing ambiguity in one of several ways; one such way being the presence of two or more deconstructors for a class which are similar and used in such a way that the compiler cannot determine which one will match at run-time.

4.3.2 Usability

The OOMatch project was not evaluated in any manner directly comparable to the testing done on the *Mach7* project. However, the way a programmer may go about using the features added by the OOMatch project is relatively intuitive for someone familiar with Java, with the only necessary changes to a class being the addition of deconstructors and the use of pattern matching being so similar to method overloading. That said, the OOMatch project is an extension to Java, and as such may require the programmer to download additional tools [5] in order to compile their code - something some would consider a hassle.

5. CONCLUSION

In conclusion, pattern matching, a convenient and highly desirable feature commonly found in functional languages, is gaining popularity to such a degree that projects have been undertaken to add these features to popular object-oriented languages such as C++ and Java. These projects, namely *Mach7* and OOMatch, do a fair job of implementing those desirable features, but do so in very different ways. In Java, with OOMatch, we see objects gaining the ability to specify how they can be decomposed and what data they will expose to whoever is decomposing them, with pattern matching used to control method dispatch. In C++, with the *Mach7* project, we see patterns composed and type-checked at compile time using expression templates, focusing on decomposition of algebraic types and with pattern matching used much in the way one would typically find a switch statement used.

Both projects seem to be in a usable state, and succeed in adding desirable features to popular languages. They are not, however, without their drawbacks. In OOMatch the programmer will need to be wary of potential pitfalls of ambiguity stemming from a decision by the authors to provide powerful, but not always safe, features. In *Mach7* a programmer will need to go through the process of learning a small bit of new syntax, as well as the extra tedium of additional setup when building their class hierarchies. The *Mach7* project was evaluated and performs well, succeeding in adding convenient features while sacrificing little to no performance. The OOMatch project does not provide any performance details. However, the code written in the OOMatch project will compile to standard JVM Bytecode, opening it up to extensive optimization by the JVM. This optimization may counteract the relatively poor performance of the *patterns-as-objects* approach demonstrated by the testing done in the *Mach7* project.

Neither project is inherently superior to the other, just as neither language is universally considered better than the other. Ultimately, the directions in which these projects proceeded with their implementations seems to follow, unsurprisingly, the style of programming popular in their respective languages.

[5] namely, Polyglot

References

[1] Curtis Clifton et al. "MultiJava: Design Rationale, Compiler Implementation, and Applications". In: *ACM Trans. Program. Lang. Syst.* 28.3 (May 2006). ISSN: 0164-0925.

[2] Haskell. *Defining Methods*. (Accessed 12-2-2016). URL: `https://wiki.haskell.org/Algebraic_data_type`.

[3] Allen Leung. *Prop: a C++ based pattern matching language*. (Accessed 12-2-2016). URL: `http://www.cs.nyu.edu/leunga/www/prop.html`.

[4] Radu Muschevici et al. "Multiple Dispatch in Practice". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008.

[5] Oracle. *Defining Methods*. (Accessed 12-2-2016). URL: `https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html`.

[6] Adam Richard and Ondrej Lhotak. "OOMatch: Pattern Matching As Dispatch in Java". In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007.

[7] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. "Open and Efficient Type Switch for C++". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. Tucson, Arizona, USA: ACM, 2012.

[8] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. "Open Pattern Matching for C++". In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. GPCE '13. Indianapolis, Indiana, USA: ACM, 2013.

[9] Wikipedia. *Multiple Dispatch*. (Accessed 12-2-2016). URL: `https://en.wikipedia.org/wiki/Multiple_dispatch`.

[10] Wikipedia. *Pattern Matching*. (Accessed 12-2-2016). URL: `https://en.wikipedia.org/wiki/Pattern_matching`.