Pattern matching in Pharo

Aless Hosry, Vincent Aranega, Nicolas Anquetil, Stéphane Ducasse

Pattern matching in Pharo

Aless Hosry[1], Vincent Aranega[1], Nicolas Anquetil[1] and Stéphane Ducasse[1]

[1]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

Pattern matching is a computational technique used to identify and analyse recurring structures or patterns within data and enable the extraction of meaningful information from the data.In the field of software engineering, pattern matching is often used in applications such as compilers and linters, where the ability to recognise and understand structural patterns in source code is essential for tasks such as optimisation, error detection, and code refactoring, but pattern matching also finds its use for querying deeply recursive structures.Pharo already offers RBParseTreeSearcher, a powerful Domain Specific Language (DSL) defining its own textual syntax with a focus on matching the Pharo Abstract Syntax Tree (AST).This tool is the base of the Pharo Refactoring Framework.However, RBParseTreeSearcher is dedicated to Pharo AST matching only and cannot perform on other kinds of structures or ASTs.In this paper, we introduce MoTion, a new pattern matcher that works on Pharo objects in the large and that focuses on flexibility and expressiveness, and then we present the syntax of both matcher tools.Finally, we compare the execution speed and the expressiveness of both pattern matchers in the context of AST matching against each other as well as against a pure Pharo request on the AST.MoTion offers a powerful base for dedicated pattern matchers and matches any kind of Pharo objects, but its genericity implies lower performances than a dedicated solution tailored for a unique usage as RBParseTreeSearcher.

Keywords

Pattern matching, Smalltalk, Object-oriented programming

1. Introduction

Pattern matching is a computational technique used to identify and analyse recurring structures or patterns within data and enables the extraction of meaningful information from the data [1].

Pattern matching could vary between string searching or regular expressions, which are supported in multiple programming languages such as Java, Javascript and C# [2], matching over trees of objects [3] like ASTs, or object matching using functional programming languages like Scala [4].

Such matchers are needed in Pharo for tasks like optimisation [5], by identifying specific patterns in code defined by developers to reduce resource consumption and enhance performance. Additionally, matchers can help in error detection by detecting common programming errors [5] and applying some code refactoring in order to improve software quality and reliability. Moreover, pattern matching is reliable for querying deeply recursive structures like nested data [6] that are very common during software analysis or data exploration in fields like data mining.

IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France

aless.hosry@inria.fr (A. Hosry); vincent.aranega@inria.fr (V. Aranega); nicolas.anquetil@inria.fr (N. Anquetil); stephane.ducasse@inria.fr (S. Ducasse)

0000-0003-1486-8399 (N. Anquetil); 0000-0001-6070-6599 (S. Ducasse)

CEUR Workshop Proceedings (CEUR-WS.org)

Based on these requirements, we present in this paper the “RBParseTreeSearcher” matcher that is already present in Pharo as well as another, more generic matcher that we developed recently and named “MoTion” [7].

The first one is the pattern matcher used by the RefactoringBrowser infrastructure [8], it allows to define patterns to match them with Pharo source code AST. The RefactoringBrowser also comes with a RBParseTreeWriter that allows to modify the source code matching a given pattern, but we will not explore this part here as we are only interested in the pattern matching capability.

The second pattern matcher is MoTion [7], a recently created generic pattern matcher that allows to express patterns on any set of objects, their classes, and their properties. The focus of MoTion is the expressiveness of patterns, offering a large set of functionalities.

In this paper, we present some of the existing approaches and how they differ from each other in Section 2. Then in sections 3 and 4 we present the syntax of the matchers followed by examples. The actual comparison takes place in section 5, before concluding the paper in Section 7.

2. Pattern matchers characteristics

Many programming languages offer pattern-matching libraries. Some of them are GPL (General Purpose Languages) like in Python, Haskell, Rust and Scala, while others are DSL (Domain Specific Languages) like with Rascal.

Table 1 presents a list of characteristics we identified in several existing implementations of pattern matching languages:

Object Matching [9] is the fact of matching any object based on its class and properties. The expected values of the properties can be themselves patterns to be matched against the actual value returned when the unary-message is called on real objects.

Structural Pattern [10] refers to a pattern that captures and matches the "shape" of a given object or data using subpatterns.

Anonymous Variables [11] refer to tokens that imply the existence of a value without capturing it.

List matching [11] offers the ability to describe the general shape of a list.

Negation [12] consists of specifying a pattern that the data should not match.

Unification [13] consists of using the same variables multiple times in a pattern that all match the same (repeated) data.

Nested Match [13] consists of expressing a pattern about a direct inner object.

Deep Match [11] allows to search for an object in a structure (or hierarchy) without having to bother about the depth to which the searched object is.

Recursive Search [4] is similar to Deep Match but assumes a hierarchy of similar objects, or at least objects that have the same property. For example, one could search for an object having a specific "child" without knowing at what depth this object, and thus its child, are.

Can yield all results [13] means that the result contains all data matching the pattern and not just the first match found.

Table 1
Existing approaches(Pattern matching languages)

Characteristics / Languages	Rascal	Haskell	Scala	Python	Java[a]	Rust
Object Matching	x	x	x	x	Matching Types	x
Structural Pattern	x	x	x	x	planned	x
Anonymous Variables	x	x	x	x	planned	x
List Matching	x					
Negation	x	x				
Unification	x					
Nested Match	x	x	x			x
Deep Match	x	x		x		x
Recursive Search		x				
Can yield all results	x					

[a]https://dev.java/learn/pattern-matching/

Many approaches in the literature applied pattern matching, whether for General Purpose Programming Languages (GPL) or Domain Specific Programming Languages (DSL) [14]. Table 1, summarises those approaches according to the characteristics described previously in this section.

3. RBParseTreeSearcher

In this section, we present the pattern matching language of the Pharo refactoring framework as implemented in RBParseTreeSearcher. We will first introduce the syntax of the language, and then we will show different usages with Pharo methods of the RBParseTreeSearcher class.

3.1. Syntax

The syntax of this language relies on specific symbols, allowing to express patterns over any source code of a Pharo image:

(‘) Back-tick. It is used to match any single node. For example: "`‘someName asString`" can match message `asString` sent to **any** receiver, disregarding its name. It can match Pharo source code like "`self asString`" and "`aParam asString`".

(‘#) Literal pattern nodes. To verify that the matched node is a literal, a back-tick can be followed by the hash sign. "`‘#lit size`" is an example where `‘#lit` can match an

integer 3, a string `foo` or even a collection such as `#(a b c)`. It will not match "`self asString`".

(‘@) List pattern nodes. Is used to match zero, one or more nodes in the AST. For example, "`|‘@args1 t1 ‘@args2|`" returns a successful match with `| t1 t2 t3 t4|` where `args1` will be empty and `args2` matches `t2 t3 t4`.

(“) Double Back-tick. It is used to perform a recursive search. This implements the *deep match* characteristic. It entails searching for patterns not only on the surface level of the source code, but also examining their internal structure. For example, "`“@vars + 1`" matches any selector followed by `+ 1`, additionally, it will internally check if this selector is already constructed by another selector followed by +1, such as `(myNum + 1) + 1 + 5`, where the first match of `‘@vars’` is `myNum + 1` and the second is `myNum` found by the deep match.

(‘.) Statement pattern nodes. To match statements, a developer can use a backtick followed by a period. Such patterns `‘.Statement1.` match a single statement in Pharo such as "`self assert: myVal size equals: 11.`". It can also be combined with the list pattern (`‘@.`) to search for a list of successive statements, such as "`‘@.statements`" which will successfully match "`x := 1. y := 2. z := OrderedCollection new.`".

(‘{ }) Block Pattern Nodes Is a free-form test where, inside the curly braces, one puts a Pharo block, receiving a node as a parameter and returning a boolean whether this node matches or not. For example: "`‘{:node|node isVariable and:[node isGlobal]} become:nil`" is a pattern that matches a message `become:` with a `nil` argument, where the receiver is a global variable.

3.2. Examples

After describing the syntax and capabilities of Pharo source code matchers over blocks, literals, statements and so on, we will go through how to use this syntax with a handful of methods defined in RBParseTreeSearcher in this section. While this paper focuses on matching in Pharo, it is important to note that the same syntax can be used with a couple of methods implemented in RBParseTreeWriter to perform some source code transformations.

Listing 1: Pattern Matching over source code sample

```
1 searcher := RBParseTreeSearcher new.
2 searcher
3     matches: '@rcv put'
4     do: [ :aNode :answer | contextDictionary := searcher context ].
5 searcher executeTree: (RBParser parseExpression: 'self put').
```

Listing 1 contains an example of source code pattern matching in Pharo. In line 1, `searcher` variable is declared as an `RBParseTreeSearcher` object. The message `#matches:do:` is sent to this variable in line 2 to start the match, where `matches:` accepts the pattern expressed using the syntax introduced in the previous section, and `do:` accepts a block of code that is

executed only when the match succeeds. On success, all matches are stored in a dictionary (context), where each key represents a part of the pattern, like "@rcv" and the value contains the matched source code. In cases where the key matches multiple source code fragments, they are encapsulated in a list and stored in the value in pairs with the convenient key. In this example, when the block is executed, it enables storing all matches of `context` in `contextDictionary`. To start matching patterns, the developer needs to use method `#executeTree:` that takes as input a parsed source code object, such as line 5, using RBParser parseExpression.

4. MoTion pattern matcher

In this section, we are going to present MoTion, a pattern matching language for objects that relies on specific symbols and a couple of methods to allow matching over objects in a Pharo image.

4.1. Syntax

Unlike the previous pattern matching language, MoTion syntax does not have the ability to explicitly represent the Pharo AST being searched for, except for literals. In simpler terms, while the old language enables the creation of patterns that directly describe the syntax of Pharo, MoTion is based on a different approach. In MoTion, representations are based on object types and, if needed, certain methods that correspond to objects in Pharo. This matching capability extends beyond Pharo language objects and includes any other object like Java objects represented in Pharo using the FamixJava metamodel.

Literal matching MoTion is able to match literals (boolean, string, symbols and numbers) by concretely expressing them like `true`, `1`, `#foo` and `'Pharo Playground'`. In order to make it more performant, MoTion also allows matching regular expressions for strings. For example: `Pharo.*ground` can match with `Pharo Playground`.

(%) Object match A basic Object match must first include the type of the Object to match as defined in Pharo such as `Color%{}`, `RBMessageNode%{}` or `FamixJavaModel%{}`.

(<=>) Properties A match could be more specific, by specifying the type of the object to match followed by the properties assigned to the specific values. A property represents a predefined method in the object class only if such methods have a return. For example: `RBComment % {#size <=> 2}` is a pattern that could match any comment that is empty, because the size of comments in Pharo must be bigger than 2 taking into consideration the double quotation marks.

(<~=>) Negation Sometimes, it may be easier to search for objects that match a pattern, which properties don't fit specific values, instead of specifying numerous properties. For that we implemented the negation symbol that helps developers to specify such patterns easily. For example: `RBComment % {#size <~=> 2}` is a pattern that could match any comment that is not empty.

(%%) Subclasses This is again a *deep match* operation. Pattern expression could be more flexible by using the double `%%` to match properties of subclasses, such as `RBLiteralNode %%{#value <=> 1}` that could match any object that extends directly or indirectly `RBLiteralNode` class.

({.}) Lists More than one property could be specified for the match, by encapsulating all properties in a list after defining the type. For example: `RBLiteralNode %%{#value <=> 1. #sourceText <=> #'a text here' }` which allows the usage of `value` and `#sourceText`.

(@) WildCards Wildcards are used to match a property associated to any value. The importance of wildcards in MoTion is their ability to save and return such values to be treated in following steps using Pharo. For example: `RBLiteralNode %%{#value <=> #'@anyValue'}`, where `@anyValue` could be anything associated to `#value`.

Composed patterns Pattern searching may require searching for complex structures, for that, MoTion allows declaring patterns constructed also of subpatterns, such as: `RBBlockNode %{ #statements <=> RBValueNode %% { #value <=> 2 } }` where `#statements` is associated to another subpattern.

Block Matcher they are used to generate new matchers dependently on the current object being matched and/or variables captured during the matching process, such as: `ObjectA %{ #lint <=> [ :collection| collection includes: 4] onCollection }` can match `objA lint: #(1 2 3 4 5 6 7)` because the list contains 4 as described by the block inside the pattern.

Conditional Matcher they are used to write complex conditional matchers dependent on the current object being matched and/or variables captured during the matching process

To make it easier to represent advanced patterns, we developed the MoTion path, which is used solely to express properties within a pattern, allowing the direct access of children's properties instead of defining subpatterns in patterns.

(>) Composed path MoTion paths are composed out of multiple properties as descendent children, which means they are not methods declared in the same class, but they are properties of the return type object. For example: `RBMessageNode % { #'selector>value' <=> #ifTrue:ifFalse: }` `#selector` is a property of `RBMessageNode` that returns an object of type `RBSelectorNode`, which in turn has a property named `value`. This helps expressing patterns in a more flexible way, by accessing directly a children's property. It is important to note, composed path allows accessing at the same time only one direct children, but the path could be extended to as many subchildren layers as the developer needs, like: `definedClasses>methodDict>ast>allChildren`.

(_) Anonymous property Anonymous properties usage is when the developer doesn't know what is the exact name of the property, or when she/he wants to access a descendant children without caring about the predecessors. In this case, multiple matches could result

if multiple properties have the same descendant children. For example: `RBMessageNode % { #'_>parent' <=> @anyParent }` could return matches for `receiver>parent` and `selector>parent` as both have `parent` as a property.

(*) Recursive search It enables searching for the same property in many children layers, specially when the developer doesn't know what is the depth of such children to precise it in a composed path. For example: `RBMethodNode % #'children*' <=> RBMessageNode % #'selector>value' <=> #ifTrue:ifFalse:` which allows searching recursively for all properties named `children` in different descendent layers, to match any one with selector #ifTrue:ifFalse:.

(as:) Matcher save The main purpose of this matcher is to encapsulate patterns and subpatterns into variables, in order to be able to retrieve their values after a successful match. For example: `RBMethodNode % #'children*' <=> RBMessageNode % #'selector>value' <=> #ifTrue:ifFalse: as: #MyMessageNode` where `#MyMessageNode` contains matching objects of type `RBMessageNode`.

4.2. Examples

Listing 2 is an example of MoTion and how we can use it to match a package. Lines 1 to 8 contain the pattern defined using MoTion syntax. Line 9 is matching this pattern with `Talents` package as it is required in MoTion using `#match:` message. The return type of this match is a `MatchingContext` which contains `isMatch` of type Boolean, and `matchingContexts` which contains a list of a bindings dictionary that in turn contains the wildcards (since their values are unknown) as keys, and the matched objects as values. However, MoTion also provides a way to retrieve only the bindings of specific wildcards such as in line 10. The return type of collectedBindings is a list of bindings, each of which has a dictionary of keys and values. If multiple matches exist, this means that multiple bindings are returned in the list.

As for the structure of the pattern, between lines 1 and 8 , it's a pattern of type `RPackage` which properties are encapsulated in a list. The first property `#classes` is associated to a wildcard `@allClasses` (line 2), to store all the classes of the matched package and retrieve them in line 10 using message `#collectBindings:for:`. The second property is constructed of a MoTion path, as we are interested in `methodDict` of `definedClasses` which in turn encapsulates a subpattern.

Listing 2: Pattern Matching over source code sample

```
1  pattern := RPackage % {
2      #classes <=> #'@allClasses'.
3      #'definedClasses>methodDict' <=> CompiledMethod % {
4          #'ast>allChildren' <=>  RBMessageNode % {
5              #'selector>value' <=> #ifTrue:ifFalse:
6          }
7      }
8  }.
9  pattern match: #Talents asPackage.
10 collectedBindings := pattern collectBindings: {#allClasses} for: (#Talents asPackage).
```

As a summary, listing 2 is a an example of a pattern to be matched with any package in Pharo, which methods contain the selector `#ifTrue:ifFalse`.

We mentioned earlier that MoTion can match objects, and we are using it to match Java, XML ans SQL Objects parsed in Pharo. Listing 3 is an example of how we can use it to retrieve XML nodes based on their attribute name and value, which cannot be done using RBParseTreeSearcher syntax.

Listing 3: Pattern Matching over source code sample

```
XMLNodeList % {
    #'_>attributes' <=> XMLAttribute % {
        #'name' <=> #'Field' .
        #'value' <=> #'myButton1'.
    }
}
```

5. Comparison

By leveraging MoTion's syntax, users can specify complex patterns to match objects based on various criteria, as we chose MoTion syntax to be expressed declaratively. In terms of capabilities, MoTion has been impacted by other matchers like the ones stated in 2 in addition to RBParseTreeSearcher such as deep search and anonymous variables. Since both matchers have some common capabilities, we decided to perform a comparison of matching using the same Pharo 11 image for both of them. We took some basic source code templates, some of which are search rules implemented as examples in the rewrite tool.

5.1. Syntax and Expressiveness

- The first example, shown in Listing 4, checks if the source code has a selector `#ifTrue:ifFalse:`. With RBParseTreeSearcher, specifying patterns to detect the receiver and the list of arguments inside the blocks is mandatory, while in MoTion, this specification could be skipped as the developer is only interested in knowing if the selector is invoked in this code or not.
- Patterns of the second example in Listing 5, are inspired by the third rule of the rewriter tool, whose purpose is to check if `nil` exists in `ifNil` to remove it by applying a transformation rule. For this rule, the RBParseTreeSearcher pattern is more efficient as it is able to precisely position nil inside `ifNil:`. This cannot be done by MoTion, as it cannot precisely determine the nil position in the used blocks.
- The third example, shown in Listing 6, is inspired by rule 8 of the rewriter tool, which consists of matching a `select:` method that contains a receiver followed by not, to be replaced in a later stage by `reject:`. Again, the precision of `not` position is mandatory in this example, which is possible by RBParseTreeSearcher as it is able to express the possible existence of temporary variables and statement lists, followed by `not` positioned at the end inside the block. While that is not possible in MoTion, as the properties declared inside the pattern and associated with some values or subpatterns, do not take into consideration their order.

Listing 4: Match #ifTrue:ifFalse:

```
"-- RBParseTreeSearcher --"
‘@receiv ifTrue: [ ‘@args1 ] ifFalse: [ ‘@args2 ].

"-- MoTion --"
RBMessageNode % {
    #’selector>value’ <=> #ifTrue:ifFalse:
}
```

Listing 5: Remove unessecary #ifNil

```
"-- RBParseTreeSearcher --"
‘@receiver ifNil: [ nil ] ifNotNil: ‘@arg

"-- MoTion --"
RBMessageNode % {
    #’selector>value’ <=>#ifNil:ifNotNil:.
    #’arguments>body>statements>value’ <=> nil
}.
```

Listing 6: Replace #select: by #reject:

```

"-- RBParseTreeSearcher --"
‘@receiver select: [ :‘each |
            | ‘@temps |
            ‘‘@.Statements.
            ‘‘@object not ]

"-- MoTion --"
RBMessageNode % {
    #’selector>value’ <=> #’select:’.
    #’children*>selector>value’ <=> ’not’
}.
```

5.2. Matching speed

Table 2
Speed comparison

Pattern	RBParseTreeSearcher Speed for 100 000 executions per second	MoTion Speed for 100 000 executions per second
Listing 4	0.101	0.433
Listing 5	0.98	1.705
Listing 6	0.411	1.389

While both languages excel in pattern matching capabilities, RBParseTreeSearcher has been recognised for its superior speed compared to MoTion as it is dedicated to matching only Pharo ASTs. It is optimised for this only, while MoTion is entirely generic and can match any object at any depth, implying more computation and thus more time to execute.

5.3. Matching characteristics

After comparing the speed of match for both languages, we list in Table 3 the characteristics explained in section 2 and if they are applied for each one of them: Based on both comparisons, it is evident that RBParseTreeSearcher exhibits superior performance compared to MoTion, despite the latter being a more generic pattern matching language.

Table 3
Characteristics comparison

Characteristics / Languages	MoTion	RBParseTreeSearcher
Object Matching	x	
Structural Pattern	x	x
Anonymous Variables	x	x
List Matching	x	x
Negation	x	
Unification	x	x
Nested Match	x	
Deep match	x	
Recursive path	x	x
Can yield all results	x	x

6. Future work

MoTion lacks for the moment two important features: a debugger and a pattern generator. Currently, we use MoTion to extract source code to do software analysis for cross-language applications, but the expressiveness of such patterns is error-prone, especially while precising the properties of a MoTion path, which sometimes leads us to lose time while expressing such patterns. At the moment, we are employing a simple technique that involves minimising MoTion paths and starting to determine whether or not we have found a match. However, a debugger will be able to automate this process and address the incorrectness of such paths and what properties may be incorrect, thus making it faster and more effective for developers to use. Additionally, we intend to develop a pattern generator that will enable developers—particularly those who are unfamiliar with MoTion—to provide it with the source code they want it to search for. MoTion will then be able to automatically generate patterns that represent such source code, enabling developers to match it against the packages or models they want to search for.

7. Conclusion

To conclude, we present in this paper two pattern matchers in Pharo. The first one, RBParseTreeSearcher syntax, is able to match Pharo AST and has the ability to apply transformations using the rewriter tool. The second one, MoTion, implemented recently for matching over objects, including Pharo ASTs, given their types and properties. Both matchers were compared, and the experiment has shown a faster performance for RBParseTreeSearcher over the same patterns, being more specific. However, we also realised that, given the flexibility of the syntax

of MoTion, a higher ability to match objects, disregarding their complexity, and is able to support structural patterns, lists, object matching, deep matching, negation, and many other characteristics compared with other tools implemented in the literature by different programming languages.

References

[1] M. Hills, Navigating the wordpress plugin landscape, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, 2016, pp. 1–10.

[2] J. E. Friedl, Mastering regular expressions, " O'Reilly Media, Inc.", 2006.

[3] C. M. Hoffmann, M. J. O'Donnell, Pattern matching in trees, Journal of the ACM (JACM) 29 (1982) 68–95.

[4] B. Emir, M. Odersky, J. Williams, Matching objects with patterns, in: ECOOP 2007–Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30-August 3, 2007. Proceedings 21, Springer, 2007, pp. 273–298.

[5] D. C. Atkinson, W. G. Griswold, Effective pattern matching of source code using abstract syntax patterns, Software: Practice and Experience 36 (2006) 413–447.

[6] M. Hills, P. Klint, J. J. Vinju, Program analysis scenarios in rascal, in: Rewriting Logic and Its Applications: 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers 9, Springer, 2012, pp. 10–30.

[7] MoTion, Motion, ???? Https://github.com/forBlindReview/MoTion.

[8] N. Anquetil, M. Campero, S. Ducasse, J.-P. Sandoval, P. Tesone, Transformation-based refactorings: a first analysis, in: IWST 22-International Workshop of Smalltalk Technologies, 2022.

[9] D. Di Ruscio, R. Eramo, A. Pierantonio, Model transformations, 2012, pp. 91–136. doi:10.1007/978-3-642-30982-3_4.

[10] L. George, A. Wider, M. Scheidgen, Type-safe model transformation languages as internal DSLs in scala, in: Theory and Practice of Model Transformations: 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings 5, Springer, 2012, pp. 160–175.

[11] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, J. Teich, An evaluation of domain-specific language technologies for code generation, in: 2014 14th International Conference on Computational Science and Its Applications, IEEE, 2014, pp. 18–26.

[12] C. Kirchner, R. Kopetz, P.-E. Moreau, Anti-pattern matching, in: Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16, Springer, 2007, pp. 110–124.

[13] Rascal, Rascal, ???? Https://www.rascal-mpl.org/docs/Rascal/Patterns/.

[14] P. Klint, T. Van Der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2009, pp. 168–177.