Accepted to IWST 2016

Phorms: Pattern Combinator Library for Pharo

Markiyan Rizun
Lviv National University of Ivan Franko,
Inria, UMR 9189 – CRIStAL
mrizun@gmail.com

Gustavo Santos
Stéphane Ducasse
Inria, UMR 9189 – CRIStAL
University of Lille, CNRS
Centrale Lille, France
{firstname.lastname}@Inria.fr

Camille Teruel
Företagsplatsen
Stockholm, Sweden
camille.teruel@gmail.com

Abstract

Pattern matching is a common mechanism to provide analysis and transformation of data structures. Such an approach basically checks whether the containing elements of a data structure are constituents of a pattern, described by the developer. This paper is a step towards having seamless object-oriented pattern matching, which would be applicable to any object in Pharo. We present a pattern matching library, called *Phorms*, which enables users to compose patterns using the syntax of the Pharo programming language. In this library, patterns are objects and therefore can be inspected and debugged using existing Pharo tools. Our solution is extensible unlike *The Rewrite Engine* – Pharo's current pattern matching facilities. Moreover, by treating patterns as first class objects, our library provides more flexibility in the pattern matching process.

Keywords pattern matching, abstract syntax tree, object-oriented programming

1. Introduction

Pattern matching is one of "most expressive features" [4] of functional programming, where it was originally introduced. The main goal of pattern matching facilities in functional programming is to facilitate definition of functions on inductive data structures. For instance, consider the following pattern matching example in OCaml.

Listing 1 shows a binary tree datatype declaration and a function flip that transforms a tree by switching the branches of each nodes. The implementation of this function completely relies on pattern matching facilities. The function is composed of two patterns: the first matches a leaf and binds its value to the variable n and the second matches a node and binds its two branches to the variables t1 and t2.

```
type btree = Leaf of int | Node of btree * btree;;

let rec flip t = match t with
  Leaf(n) -> Leaf(n)
  | Node(t1,t2) -> Node(flip t2, flip t1);;
```

Listing 1: Flipping a binary tree in OCaml

Despite the fact that the application of pattern matching facilities in Listing 1 is not appropriate for the object-oriented approach, there is no doubt that pattern matching brings great benefits to object-oriented programming languages [8, 13]. In fact, pattern matching has been seamlessly integrated into some object-oriented languages, for example in Newspeak [6] and Scala. The most common way to exploit pattern matching facilities in OOP is to decompose the object data without breaking encapsulation, *i.e.,* by not permitting the direct access to attributes of an object.

Currently, pattern matching facilities already exist in Pharo [2, 3], via the *The Rewrite Engine*, a tool that focuses on source code rewriting.[1] Since 1996, *The Rewrite Engine* [11, 12] has been widely used as an efficient support for code refactoring and code rewriting in Pharo, Dolphin Smalltalk, VisualWorks and VASmalltalk. However this tool can only express patterns of source code and not arbitrary objects, *i.e.,* patterns can only match Abstract Syntax Tree (AST) nodes.

In this paper, we propose a general pattern matching library, *i.e.,* the library is able to match not only AST nodes, but any object in Pharo. This work is inspired by the pattern matching library proposed by Geller et al. [6] for Newspeak. Our library, called *Phorms*, enables users to operate patterns as any other first-class objects, which has many advantages. Patterns can be reused and composed, new patterns and pattern combinators can be implemented by third-party libraries

[1] http://www.refactory.com/the-rewritetool

```
1 "matching part"
2 | `@temporaries |
3 `@.Statements.
4 `@receiver foo: `@argument.
5
6 "rewriting part"
7 | `@temporaries |
8 `@.Statements.
9 `@receiver bar: `@argument.
```

Listing 2: AST rewriting rule with *The Rewrite Engine*

and the standard development tools can be used to debug patterns.

The paper is organized as follows: Section 2 presents our criteria to evaluate pattern matching and how these facilities are currently proposed in Pharo. Section 3 present our solution – *Phorms*– and some examples of usage. Then, Section 4 presents a case of AST transformation using *The Rewrite Engine* and *Phorms*. We discuss our plans for future work in Section 5. Finally, we present related work in Section 6 and Section 7 concludes this paper.

2. Motivating Example

In this section, we start by discussing the desirable features of a pattern matching library specifically tailored for object-oriented programming.

These features will help us evaluate *Phorms* and compare it with other pattern matching solutions *e.g., The Rewrite Engine*. Then, we present an example of pattern matching with *The Rewrite Engine* in Section 2.2.

2.1 Requirements for a pattern matcher for OOP

In this paper, we establish the following criteria to evaluate a pattern matching library.

Composition. Composable patterns promote the reuse of existing (and potentially complex) patterns to create new ones.

Genericity. It refers to the ability of a library to match and transform any object via patterns, either created by the developer or already existing in the language.

Extensibility. A pattern matching library must be extensible, *i.e.,* one should have the possibility to specify and add custom pattern classes to the library.

Debuggability. With complex patterns, it is valuable to be able to debug the matching process to understand why a subject that is expected to be matched by a pattern is not.

2.2 Existing Solution – *The Rewrite Engine*

In this section, we provide an example of existing pattern matching in Pharo, *e.g., The Rewrite Engine*.

Listing 2 presents a code rewriting example, *i.e.,* pattern matching is used to find and transform pieces of code.

This example is made of two patterns: the first one specifies which AST nodes should be matched and the second one specifies how they should be rewritten. The first pattern matches block or method AST nodes whose last statement is a message send with selector #foo. This block or method node might have any number of temporary variables (line 2), then any number of unspecified statements (line 3) and finally a message node with an arbitrary receiver, the selector (foo:) and one arbitrary argument (line 4).

The second pattern in the rewriting part indicates how a matched node is rewritten. It keeps everything unchanged except the final message whose selector is replaced by bar:.

2.3 *The Rewrite Engine* Evaluation

The Rewrite Engine has been widely adopted since 1996 [11] as part of the Refactoring Browser. Both these tools have been available on all major Smalltalk implementations. *The Rewrite Engine* provides effective means to rewrite code. For instance, *The Rewrite Tool* [10] uses this engine to perform code rewriting.

Additionally, *The Rewrite Engine* design proposes a syntax to express patterns as close as possible to the code. As such, it introduces only a few new elements into Pharo's syntax. In the example we can clearly see that the rewrite rule is close to usual Pharo code.

Nevertheless, while *The Rewrite Engine* is a powerful tool for code transformation it has some limits.

Composition. Unfortunately, it is not possible to compose rules created with *The Rewrite Engine*.

Genericity. While one is able to effectively rewrite source code using *The Rewrite Engine*, it is not possible to match and transform any other objects than ASTs.

Extensibility. Programmers are not able to extend the functionality of *The Rewrite Engine*.

Debuggability. *The Rewrite Engine* does not provide any tools to debug or inspect rules.

3. *Phorms*

In this section, we present the main components in *Phorms* architecture, namely the *phorms*. Phorms are patterns that can be used on both sides of a rewrite rule: on the left-hand side, the *matching side*, to describe which kinds of objects are to be matched and on the right-hand side, the *transformation side*, to describe what objects are to be constructed in case of successful matches.

Each phorm type is implemented as a subclass of the Phorm class. Such classes must implement two methods: matchIn: aContext for matching and transformIn: aContext for transformation. These methods take a *context* object as argument. The context gives access to contextual information such as the *subject*, *i.e.,* object being matched, and values of bound variables from previous matches. These method can succeed, in which case they return the context passed in argument with a potentially updated subject, or they can fail, in which case they return a failure object which embeds information about where and why the match/reconstruct failed.

The behavior of the matchIn: and transformIn: is a subject to one law: if the transformIn: method succeeds, it should update the context with a subject that makes the matchIn: method succeed. That is, the responsibility of the transformIn: is to transform the subject into a new subject that the phorm matches.

In addition to pattern matching, the main superclass Phorm also provides the basic API to compose patterns. For instance, the #and: message composes two patterns into an instance of AndPhorm class, which succeeds if both sub-patterns succeed, and the #named: message wraps a pattern into an instance of NamedPhorm that binds a the subject of a successful match to a given variable name.

We describe in the following sections some of the core phorms and object deconstruction/reconstruction.

3.1 Simple Phorms

Equality phorms Equality phorms are constructed with expressions of the type *obj* equals (or equivalently *obj* asPhorm).

Used on the matching side, an equality phorm matches object equal to the given *obj* object. Used on the transformation side an equality phorm updates the subject to the given *obj* object or leaves it unchanged if the subject is already equal to *obj*.

There are also identity phorms that work similarly to identity comparison instead of equality.

Conjunctive and disjunctive phorms Conjunctive phorms are constructed with expressions of the type *p1* & *p2* & ... & *pn* and disjunctive phorms with expressions of type *p1* | *p2* | ... | *pn*).

A conjunctive phorm matches a subject if all its sub-phorms *p1,p2,...,pn* do. A disjunctive phorm matches a subject if one of sub-phorms does. These phorms are typically not used on transformation side, but they can be.

Named phorms A named phorm wraps another phorm and gives a name to successful matches for later reference in the transformation side. These phorms are created with expressions of the type *p* named: *#aName*. There is also a shorter version in case *p* is the Any phorm: *#aName* var.

Used on the matching side, a named phorm binds *#aName* to the subject if *p* succeeds and if *#aName* is not yet bound. In case *#aName* is already bound the subject must be equal to the value bound to *#aName*. This allows to express patterns where different elements of the subject must be equal. Used on the transformation side, a named phorm produces the value bound to *#aName*

List phorms List phorms are the core phorms used for object deconstruction and reconstruction.

Object deconstruction and reconstruction is performed by sending two messages to the target object: (i) #deconstruct must return a collection of attributes of the receiver (*i.e.,* the deconstructed object), and (ii) #reconstruct: must return a new instance of the same class than the receiver using an array of attributes as argument. These two messages are already implemented for Pharo's AST and for built-in Pharo classes such as Association, Collection, Point, *etc.*. For instance, Point»#deconstruct returns an array with two coordinates of this point and Point»#reconstruct: returns an instance of Point, which is created using the argument, an array with two numbers for setting x and y coordinates. Clearly, users are welcome to implement #deconstruct and #reconstruct: messages for their custom objects.

List phorms deconstruct the subject and try to match the resulting list of constituents according to its sub-phorms. A list phorm is constructed with expressions of the type *p1* , *p2* , ... , *pn*.

Used on the matching side, a list phorm succeeds if *p1* matches the first constituent of the deconstructed subject, *p2* matches the second, *etc.*. Used on the transformation side, a list phorm reconstructs the subject with an array of all its sub-phorms productions. For example the phorm 1 asPhorm, 2 asPhorm, 3 asPhorm will produce the array #(1 2 3) an use it to reconstruct the subject. The previous phorm can also be noted using an array and each element of the array is coerced to a phorm: #(1 2 3) asPhorm

While the default list phorm seen above tries to match one element of the collection with one sub-pattern, the star list pattern may match any number of elements with one sub-pattern. For example the phorm 1 asPhorm, 2 asPhorm star, 3 asPhorm matches the array #(1 2 2 2 3) as well as the array #(1 3). The star list phorm implements a *lazy* and *non-blind* matching: it will match as few objects as possible to make the entire list phorm match successfully.

Rewriting phorms Rewriting phorms wrap two other phorms to describe a transformation: first phorm is used as the matching side and the second one as the transformation side. Alternatively, the transformation side can be defined by a block, which takes the context as argument. Rewriting phorms are constructed with expressions of the type *matchPhorm* ==> *transPhorm*.

4. Evaluation

In this section we present a concrete example of AST transformation that we implement using both *The Rewrite Engine* and *Phorms*. At the end of this section we discuss and compare both approaches based on the proposed example. Listing 3 presents source code that we would like to rewrite.

```
boolVar
    ifTrue: [ var := 1 ]
    ifFalse: [ var := 2 ]
```

Listing 3: Fragment of code that we want to rewrite

Listing 4 presents what we would like to produce after rewriting. Concretely, we move the assignment to the variable var outside the message to #ifTrue:ifFalse:.

```
1 var := boolVar
2     ifTrue: [ 1 ]
3     ifFalse: [ 2 ]
```

Listing 4: Resulting code after rewriting

4.1 Matching

To begin with, we need to match the code presented in Listing 3: the use of message send #ifTrue:ifFalse with the same variable in assignment in both arguments. We also need to bind the variable to a name, to reference it in the rewriting part. Listings 5 and 6 present the matching patterns using *The Rewrite Engine* and *Phorms*, respectively.

```
1 `@condition
2     ifTrue: [ `variable := `@value1 ]
3     ifFalse: [ `variable := `@value2 ]
```

Listing 5: Pattern matching using *The Rewrite Engine*

```
1 { #condition var.
2    #ifTrue:ifFalse:.
3    { #(). #(). { { #var var. #value1 var } }.
4      #(). #(). { { #var var. #value2 var } } }
5 } asPhorm
```

Listing 6: Pattern matching using *Phorms*

Both patterns match the code in Listing 3 in a similar way.

Using *The Rewrite Engine*, the pattern definition looks similar to the code itself. We generalize both receiver of the message #ifTrue:ifFalse (line 1) and the assigned variable (lines 2 and 3). Additionally, we also generalize the assigned value to be any expression, as indicated with the expressions `value1 and `value2.

Using *Phorms*, we first specify the pattern that matches the #ifTrue:ifFalse: message. Similarly to the example with *The Rewrite Engine*, we generalize the receiver (line 1). Then we define the arguments (lines 3 and 4), two blocks that should have an assignment to the variable as statement, but no arguments nor, temporaries.

Clearly, the pattern that we implemented using *Phorms* is more complicated than the one created with *The Rewrite Engine*. Nevertheless, our solution gives more control in context of describing structure of object and its properties. *Phorms* provides a possibility for the developer to specify every detail of object in a pattern, therefore the developer will be certain of the matching results. On the other hand, the patterns created with *The Rewrite Engine* may give ambiguous results in matching.

4.2 Rewriting

Listings 7 and 8 present the code rewriting step using *The Rewrite Engine* and *Phorms*, respectively.

```
1 `variable := `@condition
2    ifTrue: [ `@value1 ]
3    ifFalse: [ `@value2 ]
```

Listing 7: Code rewriting using *The Rewrite Engine*

```
1 RBAssignmentNode of: {
2    #var var.
3    RBMessageNode of: {
4       #condition var.
5       #ifTrue:ifFalse:.
6       { RBBlockNode of: { #(). #(). #value1 var }.
7         RBBlockNode of: { #(). #(). #value2 var } }
8    }
9 }
```

Listing 8: Code rewriting using *Phorms*

Using *The Rewrite Engine*, as well as for pattern specification, the rewriting specification is closer to the source code. We specify an assignment with the variable that was inside the block (`@variable, line 1). Then we remove the assignments inside the blocks altogether. This can be compared to direct editing of usual code.

Using *Phorms*, it is necessary to reconstruct the assignment node. To accomplish that, we specify which classes will be responsible for reconstructing each AST node *e.g.,* the receiver (line 4), block (lines 6 and 7), *etc.*. Specifically when reconstructing the blocks, they will contain the assigned value. Similarly, when reconstructing the assignment node, we do it outside the message send, using the new message as argument.

Clearly, *Phorms* solution is more verbose than the syntax of *The Rewrite Engine*. However, thanks to the extensibility of *Phorms*, it is straightforward to define new composition methods (as extension-methods) on the Phorm class for ASTs equivalent to the verbose definition of the previous example. The previous matching and transformation example can be rewritten more concisely as shown in Listing 9.

```
1 "matching side"
2 #condition var
3   selector: #ifTrue:ifFalse:
4   arguments: {
5     { #var var assignTo: #value1 var } asBlock.
6     { #var var assignTo: #value2 var } asBlock }
7 ==>
8 "transformation side"
9 #var var assignTo: (
10   #condition var
11     selector: #ifTrue:ifFalse:
12     arguments: {
13        { #var var assignTo: #value1 var } asBlock.
14        { #var var assignTo: #value2 var } asBlock }
```

Listing 9: Matching and transformation using AST-specific composition methods

4.3 *Phorms* Evaluation

In this section we present advantages of our solution that it has over *The Rewrite Engine*.

Composition. In *Phorms*, the patterns are first-class objects, which allows one to compose them. On the other hand, the internal implementation of patterns in *The Rewrite Engine* does not support such functionality.

Genericity. Unlike *The Rewrite Engine*, our pattern matching facilities may be used not only to rewrite ASTs, but also to match and transform any other objects.

Extensibility. While *The Rewrite Engine* is sealed and cannot be extended, our solution enables users to add to *Phorms* their own pattern matching facilities in the form of new phorms and new composition methods (as shown in Listing 9).

Debuggability. As *Phorms* are made out of first class objects, the standard Pharo development tools like the inspector or the debugger can be used directly. Debugging patterns from *The Rewrite Engine* is a bit more tedious.

5. Future Work

To improve debugging, we plan to extend the Pharo's Inspector to generate a visualization of the pattern. This way, users would be able to inspect the structure of the pattern, *i.e.,* its sub-patterns, separately and more easily. Finally, we would like to use *Phorms* instead of *The Rewrite Engine* in our previous development – *The Rewrite Tool*.[2] This integration will allow us to transform code via *The Rewrite Tool* regardless of AST implementation.

6. Related Work

Tom is an embedded language that provides pattern matching facilities to the host language (such as *C*, *Java*, *Python*, *C++*, *C#*). It allows one to manipulate tree structures and XML documents. *Tom* has powerful matching capabilities and it is easy to use. However, pattern matching facilities are provided as syntactic constructs and not as first-class objects. The patterns are not as easily composable than with *Phorms*.

OMeta [13] is an object-oriented embedded language for pattern matching. Like *Phorms*, it is able to handle arbitrary objects. As an embedded language, it relies on a specific syntax, more concise than the one of *Phorms*. *OMeta* promotes reuse by means of inheritance between grammars, where *Phorms* promotes reuse by composition of patterns.

PetitParser [2] is a parser combinator library. Its design is close to the one of *Phorms*. Simple parsers can be composed into more complex ones by means of various combinators. However, *PetitParser* is more specifically tailored for parsing string of characters. Parsing can be seen as a special case of pattern matching focused on transforming lists (list of characters) into trees (parse trees). *Phorms* and *PetitParser* thus focus on different domains even if they functionalities overlap.

[2] Available here: http://smalltalkhub.com/#!/~MarkRizun/RewriteTool/. A video tutorial can also be found here: http://myfuncoding.blogspot.fr/

7. Conclusion

In this paper, we introduced an object-independent pattern matching library for Pharo. *Phorms* enables one to match and transform any objects, either composed by the developer or already existing in the environment. We presented examples of composition of patterns for AST transformation, in order to compare our solution with an existing library, *i.e., The Rewrite Engine*. The patterns built with *Phorms* are first-class objects, which allows one to debug and inspect them. Also, we highlight the fact that our library is easily extensible, which gives an opportunity for other developers to develop and add their own pattern matching to *Phorms*. In our evaluation, *Phorms* proved to be as efficient as *The Rewrite Engine*. We identified some challenges for future work concerning usability of our library specifically when matching and transforming the AST.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: piggybacking rewriting on java. In *Proceedings of the 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=1779782.1779787.

[2] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo*. Square Bracket Associates, 2013. ISBN 978-3-9523341-6-4. URL http://rmod.inria.fr/archives/books/Berg13a-PBE2-ESUG-2013-09-06.pdf.

[3] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. ISBN 978-3-9523341-4-0. URL http://pharobyexample.org/,http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf.

[4] P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *12th IFIP WG 6.1 International Conference and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, pages 186–200, 2010.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[6] F. Geller, R. Hirschfeld, and G. Bracha. Pattern matching for an object-oriented and dynamically typed programming language. Master's thesis, Universitätsverlag Potsdam, 2010.

[7] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[8] J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In *5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, 2003.

[9] LPM. Lazy pattern matching. http://moscova.inria.fr/ maranget/papers/warn/warn005.html.

[10] M. Rizun, J.-C. Bach, and S. Ducasse. Code transformation by direct transformation of asts. In *International Workshop on Smalltalk Technologies*, 2015. URL http://rmod.inria.fr/archives/papers/Rizu15a-CodeTransformation.pdf.

[11] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[12] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[13] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. . URL http://www.tinlizzie.org/~awarth/papers/dls07.pdf.