

Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types

SATOSHI EGI, Rakuten Institute of Technology, Japan

Pattern matching is an important feature of programming languages for data abstraction. Many pattern matching extensions have been proposed and implemented for extending the range of data types to which pattern matching is applicable. Among them, a pattern-matching system proposed by Egi and Nishiwaki features practical pattern matching for non-free data types with the following three features: (i) non-linear pattern matching with backtracking, (ii) extensibility of pattern-matching algorithms, and (iii) polymorphic patterns. They implemented their proposal in an interpreter of the Egison programming language. This paper presents a method for compiling this Egison pattern matching by introducing Scheme macros for importing Egison pattern matching into Scheme. We achieved that by transforming a matcher to a function that takes a pattern and a target, and returns lists of triples of a pattern, a matcher, and a target. This paper also demonstrates the expressiveness of this pattern-matching system by showing redefinitions of the basic list processing functions such as map, concat, or unique, and implementation of a SAT solver as a more practical mathematical algorithm.

CCS Concepts: • **Software and its engineering** → **General programming languages**;

Additional Key Words and Phrases: pattern-matching-oriented, pattern matching, non-free data types, non-linear pattern, backtracking

ACM Reference Format:

Satoshi Egi. 2019. Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types. 1, 1 (June 2019), 8 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Pattern matching is an important feature of programming languages for data abstraction. Many pattern matching extensions have been proposed and implemented for extending the range of data types to which pattern matching is applicable. (TODO: Cite) Among them, a pattern-matching system proposed by Egi and Nishiwaki features practical pattern matching for non-free data types with the following three features: (i) non-linear pattern matching with backtracking, (ii) extensibility of pattern-matching algorithms, and (iii) polymorphic patterns [8].

implementation of pattern matching for functional languages.

dynamic type system

macros

a complicate pattern matching procedure

[2]

2 RELATED WORK

We review a history of pattern matching and implementation of pattern matching in Scheme.

Author's address: Satoshi Egi, Rakuten Institute of Technology, Japan.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, or post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

XXXX-XXXX/2019/6-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

2.1 History of Pattern Matching and Its Extensions

Pattern matching that looks similar to pattern matching widely used nowadays was proposed by Burstall in 1969 [5]. Burstall proposed to use the notation “let cons(a, y) = x” instead of “let (a, y) = decons(x)”. In [5], concat is defined in the modern fashion. “:” and “::” in the second and third lines are used on behalf of “->” and “cons”, respectively.

```
1 let rec concat(xs1, xs2) = cases xs1:
2   x :: xs1: x :: concat(xs1, xs2)
3   nil() : xs2
```

User-defined algebraic data types are invented in 1970s after pattern matching. HOPE [6] by Burstall, MacQueen, and Sannella is a well-known language that introduced user-defined algebraic data types and pattern matching for them.

In 1980s, more expressive pattern matching for wider range of data types started to be pursued. Miranda’s laws [12, 13] and Wadler’s views [16] are earlier such research. They discarded the assumption that one-to-one correspondence should exist between patterns and data constructors. They enable pattern matching for data types whose data have multiple representation forms. For example, Wadler’s paper on views [16] present pattern matching for complex numbers that have two different representation forms: Cartesian and polar. However, their expressiveness is not enough for representing patterns for non-free data types. They neither support non-linear patterns and pattern matching with multiple results.

These works lead to more expressive extensions of pattern matching. Erwig’s active patterns [9] proposed in 1996 and Tullsen’s first class patterns [15] proposed in 2000 are such extensions. Both extensions allow users to customize the pattern-matching algorithm for each pattern. Active patterns supports non-linear patterns though they does not support pattern matching with multiple results. First class patterns supports pattern matching with backtracking though they does not support pattern matching with multiple results.

Egison [1] is a programming language with a pattern matching system that is extensible and supports both of non-linear patterns and multiple results [8].

```
1 (match-all (take n (repeat 0)) (multiset integer) [<cons $x <cons ,(+ x 1) _>> x])
2 ; returns [] in O(n^2) time
3 (match-all (take n (repeat 0)) (multiset integer) [<cons $x <cons ,(+ x 1) <cons ,(+ x 2) _>>> x])
4 ; returns [] in O(n^2) time
```

```
1 (define $multiset
2   (lambda [$a]
3     (matcher
4       {[<nil> []]
5        {[{} {}]}
6        [_ {}]}]
7       [<cons $ $> [a (multiset a)]
8        {[$tgt (match-all tgt (list a)
9                          [<join $hs <cons $x $ts>> [x {@hs @ts}]]]}]
10      [, $val []]
11      {[$tgt (match [val tgt] [(list a) (multiset a)]
12                    {[[<nil> <nil>] {}]}
13                    [[<cons $x $xs> <cons ,x ,xs>] {}]}]
14      [[_ _] {}]]}]
15     [$ [something]
16      {[$tgt {tgt}]]}]))
```

Pattern matching was also invented in a context of computer algebra. Pattern matching for symbolical mathematical expression was implemented in the symbol manipulation system proposed by McBride [10], which was developed on top of Lisp. This pattern-matching system supports *non-linear patterns*. A non-linear pattern is a pattern that allows multiple occurrences of the same variables in a pattern. Their paper demonstrates some examples that process symbolic mathematical expressions to show the expressive power of non-linear patterns. However, this approach does not support pattern matching with multiple results, and users cannot extend its pattern-matching facility.

At the same time, more expressive pattern matching is explored by Queinnec [11], who proposed an expressive pattern matching for lists. Though this proposal is specialized to lists and not extensible, the proposed language supports the cons and the join patterns, non-linear pattern matching with backtracking, matchAll, not-patterns, and recursive patterns. His proposal achieves almost perfect expressiveness for patterns of lists and allows the pattern-matching-oriented definition of the basic list processing functions. For example, the following member definition is presented in Queinnec's paper [11].

```
1 member ?x (??- ?x ??-) -> true
2 member ?x ?- -> false
```

2.2 Implementation History of Pattern Matching in Scheme

Currently, SRFI for pattern matching for algebraic data types does not exist. However, pattern matching for algebraic data types designed by Wright [17] is implemented in most of well-known Scheme implementations such as Gauche([TODO: Cite](#)), Guile([TODO: Cite](#)), and Chicken([TODO: Cite](#)).

Among them, Racket supports more expressive pattern matching than other Scheme implementations [14].

3 SYNTAX OF PATTERN MATCHING

$\langle pm\text{-}expr \rangle ::=$	$'(\text{match-all}' \langle expr \rangle \langle expr \rangle \langle match\text{-}clause \rangle$	(match-all)
	$ \text{'(match-first}' \langle expr \rangle \langle expr \rangle \text{'\{'} \langle match\text{-}clause \rangle^* \text{'\}'}$	(match)
$\langle match\text{-}clause \rangle ::=$	$\text{'[}' \langle pat \rangle \langle expr \rangle \text{'}]'$	(match clause)
$\langle pattern \rangle ::=$	$\text{'_}'$	(wildcard)
	$ \langle ident \rangle$	(pattern variable)
	$ \text{'\,'} \langle expr \rangle$	(value pattern)
	$ \text{'('} \langle pattern \rangle^* \text{'})'$	(tuple pattern)
	$ \text{'('} \langle ident \rangle \langle pattern \rangle^* \text{'})'$	(inductive pattern)
	$ \text{'(or}' \langle pattern \rangle^* \text{'})'$	(or-pattern)
	$ \text{'(and}' \langle pattern \rangle^* \text{'})'$	(and-pattern)
	$ \text{'(not}' \langle pattern \rangle \text{'})'$	(not-pattern)
	$ \text{'(later}' \langle pattern \rangle \text{'})'$	(later pattern)

Value patterns are represented by prepending “,” to an expression that is evaluated to a value. Tuple patterns are represented by prepending “(” to a list of patterns.

Let me show an example of a value pattern. The program below pattern-matches a list (1 2 3 2 1) as a multiset. The pattern in the program below matches if the target collection contains pairs of elements in sequence. A non-linear pattern is effectively used for expressing the pattern.

```
1 (match-all '(1 2 5 9 4) (Multiset Integer) [(cons x (cons ,(+ x 1) _)) x]) ; (1 4)
```

3.1 Tuple Patterns

Next, let me show an example of a tuple pattern. (TODO: TODO)

3.2 Logical Patterns

An or-pattern matches if one of the argument patterns matches the target.

```
1 (match-all '(1 2 3) (List Integer) [(cons (or ,1 ,10) _) "OK"]; ("OK")
```

An and-pattern matches if all the argument patterns match the target.

```
1 (match-all '(1 2 3) (List Integer) [(cons (and ,1 x) _) x]; (1)
```

A not-pattern matches if the argument pattern does not match the target.

```
1 (match-all '(1 2 3) (List Integer) [(cons x (not (cons ,x _))) x]; (1)
```

3.3 Later Patterns

A later pattern is used to change the order of the pattern-matching process. Basically, our pattern-matching system processes patterns from left to right in order. However, we sometimes want this order, for example, to refer to the value bound to the right side of pattern variables. A later pattern can be used for such purpose.

```
1 (match-all '(1 1 2 3) (List Integer) [(cons (later ,x) (cons x _)) x]; (1)
```

4 APPLICATIONS OF PATTERN MATCHING FOR NON-FREE DATA TYPES

Before introducing implementation of pattern matching, we demonstrate a pattern matching facility we are going to implement. We can execute programs shown in this section by loading a program in [2] on Gauche [3].

4.1 Redefinitions of Basic List Processing Functions

Pattern matching for non-free data types allows us to confine recursion inside a pattern and enables more intuitive definitions of even the basic list processing functions such as map, concat, and unique.

The map function is defined using pattern matching as follows.

```
1 (define pm-map
2   (lambda (f xs)
3     (match-all xs (List Something)
4       ((join _ (cons x _)) (f x))))
5
6 (pm-map (lambda (x) (+ x 10)) `(1 2 3 4))
7 ; (11 12 13 14)
```

```
1 (define pm-concat
2   (lambda (xss)
3     (match-all xss (List (List Something))
4       ((join _ (cons (join _ (cons x _)) _) x))))
5
6 (pm-concat `((1 2) (3) (4 5)))
7 ; (1 2 3 4 5)
```

```

1 (define pm-unique-simple
2   (lambda (xs)
3     (match-all xs (List Eq)
4       ((join _ (cons x (not (join _ (cons ,x _)))) x))))
5
6 (pm-unique-simple `(1 2 3 2 4))
7 ; (1 3 2 4)

```

```

1 (define pm-unique
2   (lambda (xs)
3     (match-all xs (List Eq)
4       ((join (later (not (join _ (cons ,x _)))) (cons x _) x))))
5
6 (pm-unique `(1 2 3 2 4))
7 ; (1 2 3 4)

```

4.2 Implementation of SAT Solver

The program below describes the Davis-Putnam algorithm.

```

1 (define sat
2   (lambda [vars cnf]
3     (match-first `[vars ,cnf] `[(Multiset Integer) ,(Multiset (Multiset Integer))]
4       ['[_ ()] #t]
5       ['[_ (cons () _) #f]
6         ['[_ (cons (cons 1 ()) _)
7           (sat (delete (abs 1) vars) (assign-true 1 cnf))]
8         ['[(cons v vs) (not (cons (cons ,(neg v) _) _))]
9           (sat vs (assign-true v cnf))]
10        ['[(cons v vs) (not (cons (cons ,v) _))]
11          (sat vs (assign-true (neg v) cnf))]
12        ['[(cons v vs) _]
13          (sat vs (append (resolve-on v cnf)
14                          (delete-clauses-with v (delete-clauses-with (neg v) cnf))))))])

```

4.3 Prime Numbers

match-all provided by stream-egison.scm supports pattern matching with infinitely many results.

```

1 (load "./stream-egison.scm")
2
3 (define stream-primes (stream-filter bpsw-prime? (stream-iota -1)))
4
5 (stream->list
6   (stream-take
7     (match-all stream-primes (List Integer)
8       [(join _ (cons p (cons ,(+ p 2) _))]
9         `[,p ,(+ p 2)]))
10    10))
11 ; ((3 5) (5 7) (11 13) (17 19) (29 31) (41 43) (59 61) (71 73) (101 103) (107 109))

```

```

12
13 (stream->list
14 (stream-take
15 (match-all stream-primes (List Integer)
16   [(join _ (cons p (cons (and (or ,(+ p 2) ,(+ p 4)) m) (cons ,(+ p 6) _)))]
17     `(,p ,m ,(+ p 6))])
18 8))
19 ; ((5 7 11) (7 11 13) (11 13 17) (13 17 19) (17 19 23) (37 41 43) (41 43 47) (67 71 73))

```

5 ALGORITHM OF EGISON PATTERN MATCHING

```

1 (define processMStates
2   (lambda (mStates)
3     (match mStates
4       (() '())
5       (('MState '{ ret) . rs)
6         (cons ret (processMStates rs)))
7       ((mState . rs)
8         (processMStates (append (processMState mState) rs))))))
9
10 (define processMState
11   (lambda (mState)
12     (match mState
13       (('MState {[pvar 'Something t] . mStack} ret)
14         `((MState ,mStack ,(append ret `,(t))))))
15       (('MState {[p M t] . mStack} ret)
16         (let {[next-matomss (M p t)]}
17           (map (lambda (next-matoms) `(MState ,(append next-matoms mStack) ,ret)) next-matomss))))))

```

We omit the explanation of the algorithm for enumerating infinite results of pattern matching in this paper. It is implemented in `stream-egison.scm` in [2].

6 IMPLEMENTATION METHOD

This section introduces a method for implementing the `match-all` macro in Scheme.

6.1 Compiling Matchers to lambdas

Matchers are defined as a function that takes a pattern and target, and returns next matching atoms. For example, the `Multiset` matcher is defined as follows.

```

1 (define Multiset
2   (lambda (M)
3     (lambda (p t)
4       (match p
5         (('cons px py)
6           (map (lambda (xy) `((,px ,M ,(car xy)) (,py ,(Multiset M) ,(cadr xy))))
7             (match-all t (List M)
8               [(join hs (cons x ts)) `(,x ,(append hs ts))]))))
9         (pvar
10          `(((,pvar Something ,t))))))

```

```
11      ))))
```

6.2 Compiling match-all to Application of map

The match-all expression is transformed into an application of map whose first argument uses extract-pattern-variables and second argument is the result of gen-match-results. extract-pattern-variables takes a pattern and returns a list of pattern variables that appear in the pattern. The order of the pattern variables corresponds with the order they appeared in the pattern. gen-match-results returns a list of gen-match-resultsing results. The gen-match-resultsing results consist of values that are going to bound to the pattern variables returned by extract-pattern-variables. The order of the values in the gen-match-resultsing results must correspond with the order of pattern variables returned by extract-pattern-variables.

```
1 (match-all t M [p e])
2 ;=> `(map (lambda ,(extract-pattern-variables p) ,e) ,(gen-match-results p M t))
```

The following macro do this transformation.

```
1 (define-macro (match-all t M clause)
2   (let* {[p (rewrite-pattern (list 'quasiquote (car clause))]}
3         [e (cadr clause)]]
4     `(map (lambda (ret) (apply (lambda ,(extract-pattern-variables p) ,e) ret))
5         (gen-match-results ,p ,M ,t))))
```

gen-match-results just calls processMStates introduced in Sect. 5.

```
1 (define-macro (gen-match-results p M t)
2   `(processMStates (list (list 'MState (list (list ,p ,M ,t) ) {}))))
```

6.3 Compiling value patterns to lambdas

Value patterns are transformed into lambda expressions using rewrite-pattern inside the macro. For example, “(cons x (cons y (cons ,x _)))” is transformed into “(cons x (cons y (cons (val ,(lambda (x y) x)) _)))”.

```
1 (define rewrite-pattern
2   (lambda (p)
3     (let {[ret (rewrite-pattern-helper p '())]}
4       (car ret))))
5
6 (define rewrite-pattern-helper
7   (lambda (p xs)
8     (match p
9       (('unquote q) (cons (list 'val (list 'unquote `(lambda ,xs ,q))) xs))
10      (('quote (? list? ps))
11        (let {[ret (rewrite-patterns-helper ps xs)]}
12          (cons `(quote ,(car ret)) (cdr ret))))
13      (('later p) (cons `(later ,p) xs))
14      ((c . args)
15        (let {[ret (rewrite-patterns-helper args xs)]}
16          (cons `(c . ,(car ret)) (cdr ret))))
17      ('_ (cons '_ xs))
18      (pvar (cons pvar (append xs `(,pvar))))))
```

```

19
20 (define rewrite-patterns-helper
21   (lambda (ps xs)
22     (match ps
23       ((()) (cons '() xs))
24       ((p . qs)
25        (let* {[ret (rewrite-pattern-helper p xs)]
26               [p2 (car ret)]
27               [xs2 (cdr ret)]
28               [ret2 (rewrite-patterns-helper qs xs2)]
29               [qs2 (car ret2)]
30               [ys (cdr ret2)]}]
31          (cons (cons p2 qs2) ys))))))

```

7 FUTURE WORK

7.1 Performance

7.2 Importation of More Expressive Pattern Constructs

Extending our macros for supporting loop patterns [7] is also future work.

7.3 Integration with Other Languages

Integration with a static type system remains as future work. Currently, we have an experimental Template Haskell implementation [4].

8 CONCLUSION

(TODO: Write something cool.)

REFERENCES

- [1] 2011. The Egison Programming Language. <https://www.egison.org>. [Online; accessed 14-June-2018].
- [2] 2018. egison/egison-scheme: Scheme Macros for Egison Pattern Matching. [Online; accessed 21-Feb-2019].
- [3] 2018. Gauche – A Scheme Implementation. <http://practical-scheme.net/gauche>. [Online; accessed 1-June-2019].
- [4] 2019. egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. [Online; accessed 21-Feb-2019].
- [5] Rod M. Burstall. 1969. Proving properties of programs by structural induction. *Comput. J.* 12, 1 (1969), 41–48.
- [6] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*. ACM, 136–143.
- [7] Satoshi Egi. 2018. Loop Patterns: Extension of Kleene Star Operator for More Expressive Pattern Matching against Arbitrary Data Structures. *The Scheme and Functional Programming Workshop*.
- [8] Satoshi Egi and Yuichi Nishiwaki. 2018. Non-linear Pattern Matching with Backtracking for Non-free Data Types. In *Asian Symposium on Programming Languages and Systems*. Springer, 3–23.
- [9] Martin Erwig. 1996. Active patterns. *Implementation of Functional Languages* (1996).
- [10] F. V. McBride, D. J. T. Morrison, and R. M. Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969).
- [11] Christian Queinnec. 1990. Compilation of non-linear, second order patterns on S-expressions. In *Programming Language Implementation and Logic Programming*.
- [12] Simon Thompson. 1986. Laws in miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming*.
- [13] Simon Thompson. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming* 13, 2-3 (1990).
- [14] Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).
- [15] Mark Tullsen. 2000. First Class Patterns. *Practical Aspects of Declarative Languages*.
- [16] Philip Wadler. 1987. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*.
- [17] A.K. Wright and B.F. Duba. 1993. Pattern matching for Scheme. *Unpublished manuscript* (1993).