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

SATOSHI EGI, Rakuten Institute of Technology, Rakuten, Inc. and University of Tokyo, 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, the pattern-matching system proposed by Egi and Nishiwaki features practical pattern matching for non-free data types by providing an extensible non-linear pattern-matching facility with backtracking [20]. However, they implemented their proposal only in an interpreter of the Egison programming language. This paper presents a method for compiling this Egison pattern matching by introducing Scheme macros. This method is based on the three key ideas: (i) transformation of 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, (ii) compilation of `match-all` to application of the `map` function, and (iii) transformation of a value pattern to a function that takes an intermediate pattern-matching result and returns a value. This paper also presents benchmark results that show Egison pattern-matching embedded in Gauche Scheme is faster than the original Egison interpreter.

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

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

1 INTRODUCTION

Pattern matching is an important feature of programming languages for data abstraction. Pattern matching allows us to replace many verbose function applications for decomposing data (such as `car` and `cdr`) to an intuitive pattern. The pattern-matching facility for algebraic data types is common in modern functional programming languages. For example, Haskell and OCaml support pattern matching for algebraic data types. Many Scheme implementations also have a similar pattern-matching facility [8] though Scheme does not contain pattern matching for algebraic data types in its specification, R7RS [29].

However, some data types are not algebraic data types and we cannot use the common pattern-matching facility for them. These data types are called *non-free* data types. Non-free data have no canonical form as data of algebraic data types. For example, multisets are non-free data types because the multiset $\{a, b, b\}$ has two other equivalent but syntactically different forms $\{b, a, b\}$ and $\{b, b, a\}$.

Many pattern-matching extensions have been proposed for extending the range of data types to which pattern matching is applicable to non-free data types [25, 35]. Among them, the pattern-matching system proposed by Egi and Nishiwaki [20] 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. However, the pattern-matching algorithm proposed by them is not as simple as the earlier pattern-matching extensions that can be compiled into simple conditional branches and Egi and Nishiwaki implemented their proposal only in an interpreter of the Egison programming language. As a result, a method for importing their pattern matching system into the other functional programming languages was not obvious.

This paper presents a method for integrating this pattern-matching facility of Egison to a dynamically typed functional programming language by introducing Scheme macros. These macros have been already open-sourced [2]. We can try them on the Gauche Scheme [3].

The remainder of this paper is organized as follows. Sect. 2 reviews the history of pattern-matching extensions and implementation of pattern matching in Scheme. Sect. 3 introduces the syntax of the proposed Scheme macros for pattern matching. Sect. 4 explains the pattern matching algorithm of Egison and implementation

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

of this algorithm in Scheme. Sect. 5 explains the implementation of the proposed macros. Sect. 6 discusses the performance of the proposed macros. Sect. 7 mentions the remained work. Finally, Sect. 8 concludes the paper.

2 RELATED WORK

This section reviews the history of pattern matching and implementation of pattern matching in Scheme.

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 [17]. Burstall proposed to use the notation "`let cons(a, y) = x`" instead of "`let (a, y) = decons(x)`". In [17], `concat` is defined in the modern fashion. Algebraic data types were introduced after pattern matching. HOPE [18] proposed in 1980 by Burstall, MacQueen, and Sannella is a well-known language that introduced user-defined algebraic data types and pattern matching for them.

In the 1980s, more expressive pattern matching for a wider range of data types started to be pursued. Miranda's laws [31, 32] and Wadler's views [36] are early 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 [36] presents pattern matching for complex numbers that have two different representation forms: Cartesian and polar. However, the expressive power of these pattern-matching systems is not enough for representing patterns for non-free data types. These pattern-matching systems support neither non-linear patterns nor pattern matching with multiple results.

These works in 1980s lead to the development of more pattern-matching extensions. Erwig's active patterns [21] proposed in 1996 and Tullsen's first class patterns [34] proposed in 2000 are such extensions. Both extensions allow users to customize the pattern-matching algorithm for each pattern constructor. Active patterns support non-linear patterns though they do not support pattern matching with multiple results. First class patterns support pattern matching with backtracking though they do 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 [20]. The expressions below match a list "`{1 2 3}`" as a list of integers "`(list integer)`" using the pattern "`<cons $x $ts>`". In Egison, a pattern variable begin with "`$`". Egison uses four kinds of parenthesis for different purpose. "`()`" is used to build a syntax tree or apply a function. "`<>`" is used to apply a pattern constructor to patterns. "`[]`" is used to build a tuple. "`{}`" is used to build a list.

```
(match-all {1 2 3} (list integer) [<cons $x $ts> [x ts]]) ; {[1 {2 3}]}
```

The `match-all` expression used in the above program returns a list of the results evaluating the body expression for all the pattern-matching results. The `cons` pattern decomposes a list into the head element and the rest. Therefore, in this case, the evaluation result of this `match-all` contains just a single element. The pattern-matching expression of Egison takes a *matcher* in addition to a target and match clauses. A matcher specifies the method to interpret the pattern. The matcher "`(list integer)`" specifies that we pattern-match the target with the given pattern as a list of integers. Users can define their own matchers in Egison.

If we change the matcher of the above `match-all` from "`(list integer)`" to "`(multiset integer)`", the evaluation result also changes. The `cons` pattern of a multiset is defined to divide a target list into an arbitrary element and the rest. As a result, there are three decompositions.

```
(match-all {1 2 3} (multiset integer) [<cons $x $ts> [x ts]])
; {[1 {2 3}] [2 {1 3}] [3 {1 2}]}
```

In addition to `cons`, the `join` pattern can be used for the `list` matcher. The `join` pattern divides a target list into two lists, a head part and tail part.

```
(match-all {1 2 3} (list integer) [<join $hs $ts> [hs ts]])
; {[{} {1 2 3}] [{1} {2 3}] [{1 2} {3}] [{1 2 3} {}]}
```

The expressions below match a list that consists of n zeros as a multiset of integers with patterns that match a sequential triple "`<cons $x <cons ,(+ x 1) <cons ,(+ x 2) _>>>`". In the pattern-matching expression below, the target list does not contain a sequential triple. As a result, this expression returns an empty list. In Egison, a value pattern that checks equality of data begins with "`,`". After "`,`", we can write an arbitrary expression. The time complexity of this pattern matching is $O(n^2)$ because Egison uses backtracking for traversing a search tree. The Curry functional logic programming language also supports non-linear pattern matching with multiple results. However, Curry transforms a non-linear pattern to pattern guards [14, 15, 23]. Therefore, the time complexity for the same pattern matching is $O(n^3)$ in Curry because pattern guards are applied after pattern matching succeeds. Thus, backtracking is important feature for handling non-linear patterns.

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

Of course, if we use a more efficient data structure for lists, we can find a sequential triple more efficiently. For example, if we assume that the target list is sorted, the time complexity for finding a sequential triple is $O(n)$. In fact, we can also define such a pattern-matching algorithm by defining a matcher for sorted-lists.

The program below defines a pattern-matching algorithm for a multiset.

```
(define $multiset
  (lambda [$a]
    (matcher
      {[<nil> []
        {[{} {[]}]
         [_ {}]}]
       [<cons $ $> [a (multiset a)]
        {[$tgt (match-all tgt (list a)
                 [<join $hs <cons $x $ts>> [x {@hs @ts}]])]}]
       [,$val []
        {[$tgt (match [val tgt] [(list a) (multiset a)]
                 {[[<nil> <nil>] {[]}]
                  [[<cons $x $xs> <cons ,x ,xs>] {[]}]
                  [[_ _] {}]])]}]
       [$ [something]
        {[$tgt {tgt}]}]})))
```

The `matcher` expression used in the above program is a built-in syntax of Egison for defining pattern-matching algorithms for each data type. The definition of `multiset` in Scheme will be explained in Sect. 5.1. I will skip the explanation of the following program for now. This paper imports the pattern-matching facility of Egison into Scheme.

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 [26], which was developed on top of Lisp. This pattern-matching system supports non-linear patterns. Their paper demonstrates some examples that process symbolical 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.

Queinnec [28] also pursued expressive pattern matching. Though this proposal is specialized to lists and not extensible, the proposed pattern-matching system is as expressive as Egison. The proposed system supports the `cons` and the `join` patterns, `match-all`, not-patterns, and recursive patterns.

2.2 Implementation History of Pattern Matching in Scheme

Currently, a SRFI for pattern matching for algebraic data types does not exist. However, pattern matching for algebraic data types designed by Wright [37] is implemented in most of the well-known Scheme implementations such as Gauche [8], Guile [12], and Chicken [13]. Among them, Racket [22] provides more expressive pattern matching than other Scheme implementations. The `match` expander [33] of Racket allows users arbitrary transformation of data when pattern matching. For example, we can describe view patterns [36] using the `match` expander.

This paper implements more expressive pattern matching that supports non-linear pattern matching with backtracking just by introducing Scheme macros. This work can be easily ported to other relatives of Scheme. For example, Yuito Murase has exported the proposed Scheme macros to Common Lisp [4].

3 SYNTAX OF THE PROPOSED MACROS

This section introduces the usage of the pattern-matching expressions provided by the proposed Scheme macros. Fig. 1 shows the formal grammar of the pattern-matching expressions of the proposed Scheme macros. M denotes an expression for pattern matching. e denotes an arbitrary expression including M. p denotes a pattern. x denotes a symbol. C denotes a pattern constructor.

$$
\begin{aligned}
M ::= & \texttt{(match-all } e\ e\ \texttt{[}p\ e\texttt{]} \cdots\texttt{)} \\
& \mid \texttt{(match-first } e\ e\ \texttt{[}p\ e\texttt{]} \cdots\texttt{)} \\
& \mid \texttt{Something}
\end{aligned}
\qquad
\begin{aligned}
p ::= & \ \texttt{\_} \mid x \mid \texttt{,}e \mid \texttt{(}C\ p \cdots\texttt{)} \mid \texttt{'(}p \cdots\texttt{)} \\
& \mid \texttt{(or } p \cdots\texttt{)} \mid \texttt{(and } p \cdots\texttt{)} \mid \texttt{(not } p\texttt{)} \\
& \mid \texttt{(later } p\texttt{)}
\end{aligned}
$$

Fig. 1. Syntax of the proposed macros

Scheme programs in this paper use three kinds of parenthesis. "`[]`" is used to build a tuple. "`{}`" is used to build a list. "`()`" is used to build a syntax tree or apply a function and a pattern constructor. Gauche treats these three parenthesis in the same way as "`()`".(TODO: corect?) Though we use "<>" for applying a pattern constructor in Egison, we use "`()`" for that in Scheme because we cannot use "<>" as parenthesis in Scheme. [1]

The `match-all` expression has the completely same meaning as that of Egison. The `match-first` is similar to the traditional `match` expression; it evaluates the body of the first match clause whose pattern matches with the target. We do not use the name `match` to avoid the name conflict with Wright's `match` [37] because Wright's `match` plays fundamental role for defining a user-defined matcher, which will be explained in Sect. 5.1. The only difference between `match-first` and the traditional `match` expression is `match-first` takes a matcher. The both `match-all` and `match-first` expression can take multiple match clauses. `Something` is the only built-in matcher of the Egison pattern-matching system. `Something` can handle only wildcards or pattern variables, and is the only matcher that can bind a value to a pattern variable.

The proposed macros provide several pattern constructs. The rest of this subsection explains each of them.

[1] The square brackets "`[]`" are reserved for future language extension in R7RS. Furthermore, the use of braces "`{}`" is outside the standard language specification in both of R6RS and R7RS. Therefore, it is safe to replace braces and square brackets to parenthesis when we export the Gauche Scheme program in this paper to other Scheme implementation.

3.1 Wildcard and Pattern Variables

Symbols that appear in a pattern are handled as pattern variables. The value assigned to a pattern variable can be referred to in its right side of the pattern. "_" represents a wildcard. A wildcard is pattern-matched with a target in the same method with a pattern variable though no assignment is created for a wildcard.

3.2 Value Patterns

Value patterns are represented by prepending "," to an expression that is evaluated to a value. Value patterns are used for expressing non-linear patterns. The pattern-matching algorithms for value patterns are also defined in matchers.

The program below pattern-matches a list `(1 2 5 9 4)` as a multiset. "`,(+ x 1)`" inside the pattern is a value pattern. The value assigned to the pattern variable `x` appeared in the left side of the pattern can be referred to inside the value pattern. The pattern in the program below matches if the target collection contains pairs of elements in a sequence.

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

The use of "," to notate a value pattern is an analogy of the use of "," in `quasiquote`. A value pattern is similar to `unquote` in the sense that only the expressions inside value patterns that are followed after "," are evaluated as an ordinary Scheme expression in a pattern.

3.3 Inductive Patterns

When the pattern is a list, the first element is always handled as a pattern constructor. For example, `join` and `cons` that appear in "`(join _ (cons x _))`" are pattern constructors. The pattern-matching algorithm for handling them is defined and retained in matchers. For example, in the `List` matcher, `join` is defined as a pattern constructor that divides a collection into a head and tail part, and `cons` is defined as a pattern constructor that divides a collection into the head element and rest. As a result, the following pattern matches each element of the target list.

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

3.4 Tuple Patterns

Tuple patterns are represented by prepending "`'`" to a list of patterns. Each element of a tuple pattern is pattern-matched with the corresponding element of a target list using the corresponding element of a matcher list as a matcher.

```
(match-all '[1 2] `[,Integer ,Integer] ['[x y] `(,x ,y)]) ; ((1 2))
(match-all '[1 2 3] `[,Integer ,Integer ,Integer] ['[x y z] `(,x ,y ,z)]) ; ((1 2 3))
```

"`'`" is important to distinguish a tuple pattern from a inductive pattern. For example, "`'[x y]`" cannot be distinguished from an inductive pattern whose constructor is "x" if "`'`" is not prepended.

3.5 Logical Patterns

Logical pattern constructs, or-patterns, and-patterns, and not-patterns are useful also in Egison. Its usage is similar to that in the existing languages. An or-pattern matches if one of the argument patterns matches the target. An and-pattern matches if all the argument patterns match the target. A not-pattern matches if the argument pattern does not match the target.

```
(match-all '(1 2 3) (List Integer) [(cons (or ,1 ,10) _) "OK"]) ; ("OK")
(match-all '(1 2 3) (List Integer) [(cons (and ,1 x) _) x]) ; (1)
(match-all '(1 2 3) (List Integer) [(cons x (not (cons ,x _))) x]) ; (1)
```

3.6 Later Patterns

A later pattern is used to change the order of the pattern-matching process. Basically, the pattern-matching system of Egison processes patterns from left to right in order. However, we sometimes want to change 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. The pattern inside a later pattern is pattern-matched after pattern matching for the other parts of the pattern has succeeded.

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

4 ALGORITHM OF EGISON PATTERN MATCHING AND ITS IMPLEMENTATION IN SCHEME

This section explains and implements the pattern-matching algorithm of Egison, which is described in detail in Sect. 5 and Sect. 7 of the original paper of Egison [20]. After the explanation of the pattern-matching algorithm, I introduce a simplified implementation of this algorithm in Scheme. This implementation is called by the proposed macros whose implementation are introduced in Sect. 5.

First, I start by explaining the pattern-matching algorithm of Egison briefly. In Egison, pattern matching is implemented as reductions of *matching states*. A matching state consists of a stack of *matching atoms* and an intermediate result of pattern matching. A matching atom is a triple of a pattern, target, and matcher. In each step of a pattern-matching process, the top matching atom of the stack of matching atoms is popped off. From this matching atom, a list of lists of next matching atoms is calculated. Each list of the next matching atoms is pushed on the stack of matching atoms of the matching state. As a result, a single matching state is reduced to multiple stacks of matching states in a single reduction step. Pattern matching is recursively executed for each matching state. When a stack becomes empty, it means pattern matching for this matching state succeeded. Fig. 2 shows the reduction path when executing the `match-all` expression below. The matching state that is not grayed-out in the i-th row is reduced to matching states in the $(i + 1)$-th row. For example, the first matching state in the second row is reduced to the matching state in the third row. `MState` takes two arguments, a stack of matching atoms and an intermediate pattern-matching result. The value pattern `,m` is transformed to `(val ,(lambda (m) m))`. This transformation is explained in Sect. 5.3.

```
(match-all '(2 8 2) (Multiset Integer) [(cons m (cons ,m _)) m]) ; (2 2)
```

The `processMState` function in Fig. 3a implements this process. `processMState` takes a matching state as its argument and returns a list of next matching states. In the program in Fig. 3a, several matching clauses are omitted for simplification. "`...`" in the 4th line represents this omission.

The match clause in the 5th and 6th lines describes the case where the matcher of the top matching atom is `Something`. In this case, the value of the target "`t`" is added to the intermediate pattern-matching result. The `Something` matcher is simply defined as follows.

```
(define Something 'Something)
```

The match clause in the 7th-9th lines describes the general case. The list of lists of next matching atoms is calculated in the 8th line. In this Scheme implementation, a matcher is a function that takes a pattern and a target, and returns a list of lists of the next matching atoms as explained again in Sect. 5.1.

```
1  (MState {[(cons m (cons (val ,(lambda (m) m)) _)) (Multiset Integer) {2 8 2}]} {})
2  (MState {[m Integer 2] [(cons (val ,(lambda (m) m)) _) (Multiset Integer) {8 2}]} {})
   (MState {[m Integer 8] [(cons (val ,(lambda (m) m)) _) (Multiset Integer) {2 2}]} {})
   (MState {[m Integer 2] [(cons (val ,(lambda (m) m)) _) (Multiset Integer) {2 8}]} {})
3  (MState {[m Something 2] [(cons (val ,(lambda (m) m)) _) (Multiset Integer) {8 2}]} {})
4  (MState {[(cons (val ,(lambda (m) m)) _) (Multiset Integer) {8 2}]} {[m 2]})
5  (MState {[,m Integer 8] [_(Multiset Integer) {2}]} {2})
   (MState {[,m Integer 2] [_(Multiset Integer) {8}]} {2})
6  (MState {[_(Multiset Integer) {8}]} {2})
7  (MState {[_Something {8}]} {2})
8  (MState {} {2})
```

Fig. 2. Reduction path of matching states

In the program in Fig. 3a, the matching clauses for and-patterns, or-patterns, not-patterns, and later patterns introduced in Sect. 3 are omitted. We can implement these pattern constructs by adding match clauses for them. We can see the full implementation of `processMState` in `egison.scm` in the GitHub repository of the proposed macros [2].

The `processMStates` in Fig. 3b implements the whole pattern-matching process while the above `processMState` implements a step of this pattern-matching procedure. `processMStates` takes a list of matching states as its argument and returns all pattern-matching results. The first match clause (line 4) describes the case where a list of matching states is empty. In this case, `processMStates` returns an empty list and terminates the pattern-matching procedure. The second match clause (lines 5 and 6) describes the case where the stack of the first matching state is empty. In this case, the intermediate pattern-matching result of this matching state is added to the return value of `processMStates` as a final result of pattern matching. `processMStates` is recursively called for the remaining matching states. The third match clause (lines 7 and 8) describes the general case. In this case, `processMState` is called for the first matching state. `processMStates` is recursively called for the result of appending the result of this `processMState` and the remaining matching states.

The above implementation is from `egison.scm` [2] that does not support pattern matching with infinitely many results. In `processMStates` of `stream-egison.scm` [2], the traversal of an infinitely large search tree is implemented that is explained in Sect. 5.2 of the original Egison paper [20]. Appendix B demonstrates pattern matching that uses the macros provided by `stream-egison.scm`.

`processMStates` strictly evaluates all the pattern-matching results even when only the first pattern-matching result is used. Consequently, sharing the same `processMStates` function with `match-first` is inefficient. For that reason, the `processMStates1` function in Fig. 3c is used for `match-first` instead of `processMStates`. `processMStates1` is defined to return only the first pattern-matching result. This distinct implementation of `processMStates` is not necessary for the Egison language that adopt a lazy evaluation strategy.

5 IMPLEMENTATION METHOD

There are three problems for porting the Egison pattern-matching system into Scheme. The three tricks explained in Sect. 5.1, Sect. 5.2, and Sect. 5.3 solve these problems, respectively.

(1) How to represent a matcher in Scheme that is a special built-in object of Egison?
(2) How to implement a syntax construct that calls the complicated pattern-matching procedure in Sect. 4?
(3) How to represent a non-linear pattern that has the complex scoping rule in Scheme?

```
(define processMState
  (lambda (mState)
    (match mState
      ...
      [('MState {[pvar 'Something t] . mStack} ret)
       `((MState ,mStack ,(append ret `{,t})))]
      [('MState {[p M t] . mStack} ret)
       (let {[next-matomss (M p t)]}
         (map (lambda (next-matoms) `(MState ,(append next-matoms mStack) ,ret)) next-matomss))])))
```

(a) The processMState function

```
(define processMStates
  (lambda (mStates)
    (match mStates
      [() '{}]
      [(('MState '{} ret) . rs)
       (cons ret (processMStates rs))]
      [(mState . rs)
       (processMStates (append (processMState mState) rs))])))
```

(b) The processMStates function

```
(define processMStates1
  (lambda (mStates)
    (match mStates
      [() '{}]
      [(('MState '{} ret) . rs)
       `(,ret)]
      [(mState . rs)
       (processMStates1 (append (processMState mState) rs))])))
```

(c) The processMStates1 function

Fig. 3. Implementation of the Egison pattern matching procedure in Scheme

5.1 Compiling Matchers to lambdas

In the Egison pattern matching system [20], a matcher is a special object that retains a pattern-matching algorithm. In the proposed Scheme macros, we encode a matcher with lambdas to avoid introducing new built-in objects. We can achieve that by defining a matcher as a function that takes a pattern and a target, and returns a list of lists of next matching atoms. For example, the Multiset matcher is defined as follows.

```
(define Multiset
  (lambda (M)
    (lambda (p t)
      (match p
        [('nil) (if (eq? t '{}) '{{}} '{})]
        [('cons px py)
         (map (lambda (xy) `{[,px ,M ,(car xy)] [,py ,(Multiset M) ,(cadr xy)]})
              (match-all t (List M)
```

```
9                [(join hs (cons x ts)) `[,x ,(append hs ts)]])]
10         [('val v)
11          (match-first `[,t ,v] `[,(List M) ,(Multiset M)]
12            ['[(nil) (nil)] '{{}}]
13            ['[(cons x xs) (cons ,x ,xs)] '{{}}]
14            ['[_ _] '{})]
15         [pvar `{{[,pvar Something ,t]}}])))
```

Note that the separate use of "`{}`" and "`[]`" introduced in Sect. 3 enhances the readability of the above program.

As a matcher definition of Egison [20], a matcher definition consists of match clauses for pattern matching against a pattern. Each match clause describes a pattern-matching algorithm against the matched pattern. For pattern matching against patterns, the traditional Wright's `match` expression [37] is used.

The first match clause (line 5) describes the pattern-matching algorithm for the `nil` pattern that matches with an empty list. When the target list is empty, this match clause returns a list that consists of an empty list of matching atoms. Otherwise, this match clause returns an empty list. That means pattern matching fails.

The second match clause (lines 6-9) describes a pattern-matching algorithm for the `cons` pattern. The pattern of this match clause is "`('cons px py)`". The first and second argument of the `cons` pattern is assigned to `px` and `py` respectively. The body of this match clause is a bit complicated. The evaluation result of this body expression depends on the target list. Now, we consider a case where the target list is "`(1 2 3)`". In this case, this body expression is evaluated to "`{{[px M 1] [py (Multiset M) (2 3)]} {[px M 2] [py (Multiset M) (1 3)]} {[px M 3] [py (Multiset M) (1 2)]}}`". Each list of the next matching atoms is pushed to the current matching state; as a result, three new matching states are generated. For the first matching state, `1` and `(2 3)` are pattern-matched using the "`M`" and "`(Multiset M)`" matcher with the patterns `px` and `py`, respectively, in the succeeding pattern-matching process.

The third match clause (lines 10-14) describes a pattern-matching algorithm for a value pattern. The body of this match clause compares the target and the value inside the value pattern. The `match-first` expression that recursively calls the `Multiset` matcher is used for this comparison.

The fourth match clause (line 15) describes a pattern-matching algorithm for a pattern variable and wildcard. This match clause creates the next matching atom by just changing the matcher from "`(Multiset M)`" to `Something`.

5.2 Compiling `match-all` to an Application of `map`

The `match-all` expression is transformed into an application of `map` whose first argument is a function whose argument is a return value of `extract-pattern-variables`, and second argument is the result of `gen-match-results`. The `extract-pattern-variables` function 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. For example, "`(extract-pattern-variables '(cons x xs)`" returns "`(x xs)`". The `gen-match-results` function takes a pattern, a target, and a matcher, and returns a list of pattern-matching results. These pattern-matching results consist of values that are going to be bound to the pattern variables returned by `extract-pattern-variables`. The order of the values in the `gen-match-results` must correspond with the order of pattern variables returned by `extract-pattern-variables`. For example, "`(gen-match-results '(cons x xs) (Multiset Something) '(1 2 3))`" returns "`((1 (2 3)) (2 (1 3)) (3 (1 2)))`".

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

The above transformation is done by the following macro.

```
1 (define-macro (match-all t M . clauses)
```

```
  (if (eq? clauses '())
      '()
      (let* {[clause (car clauses)]
             [p (rewrite-pattern (list 'quasiquote (car clause)))]
             [es (cdr clause)]}
        `(append (map (lambda (ret) (apply (lambda ,(extract-pattern-variables p) . ,es) ret))
                      (gen-match-results ,p ,M ,t))
                 (match-all ,t ,M . ,(cdr clauses))))))
```

`define-macro` in the above program is a traditional non-hygienic macro similar to `defmacro` of Common Lisp [11]. The `match-all` macro requires to call the complex Scheme procedures such as `extract-pattern-variables` for expanding the macro. Therefore, defining the macro just by using `syntax-rules`, which is supported in R7RS [29] is difficult. By using `syntax-case` and `syntax->datum` in R6RS [30], we can define the above macro relatively easily.

The program below defines `gen-match-results`. `gen-match-results` is a function that calls the `processMStates` function introduced in Sect. 4. An initial matching state "`(MState {[p M t]} {})`" is created from "`p`", "`M`", and "`t`".

```
(define gen-match-results
  (lambda (p M t)
    (processMStates `{(MState {[,p ,M ,t]} {})})))
```

Generally, pattern matching for algebraic data types is compiled to an expression that uses only more primitive conditional branches such as the `if` expressions [27]. As a result, the compiled program is as efficient as the program that was written manually without using pattern matching. However, the pattern-matching procedure of Egison is so complicated that it is impossible to compile pattern-matching directly to an expression only with more primitive conditional branches. This section showed that we can compile the large pattern-matching procedure of Egison by separating the extraction of pattern variables and the calculation of pattern-matching results. These two procedures are implemented in Scheme and a pattern-matching expression is compiled to a program that calls these two procedures. This method allows us to compile very complicated pattern matching procedures but has overhead. Sect. 6.2 analyzes this overhead.

5.3 Compiling Value Patterns to `lambda`s

This section explains a method for compiling value patterns. A value pattern is transformed into a `lambda` expression whose arguments are pattern variables that appear in the left side of the value pattern. For example, "`(cons x (cons y (cons ,x (cons z _))))`" is transformed into "`(cons x (cons y (cons (val ,(lambda (x y) x)) (cons z _))))`". The `rewrite-pattern` function called inside the macro does this transformation.

This transformed value pattern is handled in `processMState`. The following program shows a match clause in `processMState` for handling a value pattern, which is omitted in the explanation of `processMState` in Sect. 4.

```
(define processMState
  (lambda (mState)
    (match mState
      [('MState {[('val f) M t] . mStack} ret)
       (let {[next-matomss (M `(val ,(apply f ret)) t)]}
         (map (lambda (next-matoms) `(MState ,(append next-matoms mStack) ,ret)) next-matomss))]
      ...)))
```

When the pattern of the top matching atom is a value pattern, it applies the intermediate pattern-matching result to the function in the value pattern and passes it to the matcher. This pattern is handled by the third match clause of the `Multiset` matcher in Sect. 5.1, for example.

6 PERFORMANCE

The section discusses the performance of the proposed Scheme macros. Sect. 6.1 presents an idea for improving the performance of Egison pattern matching that was implemented for the first time in the proposed Scheme macros. Sect. 6.2 shows the benchmark results of Egison and the proposed Scheme macros and analyzes these benchmark results.

6.1 Tips for Improving Performance

There are cases that the performance of pattern matching improves by adding proper match clauses to a matcher. This subsection introduces such examples.

First, we can improve the performance of the `Multiset` implementation in Sect. 5.1 using this method. The `Multiset` matcher definition below has a new match clause in the 6th line. This match clause describes a pattern-matching algorithm for the `cons` pattern whose second argument is a wildcard. In this case, we do not need to calculate the rest of the collection. The cost for calculating the rest of the collection is so high that the performance dramatically improves by adding this new match clause.

```
1  (define Multiset
2    (lambda (M)
3      (lambda (p t)
4        (match p
5          ...
6          [('cons px '_) (map (lambda (x) `{[,px ,M ,x]}) t)]
7          [('cons px py)
8           (map (lambda (xy) `{[,px ,M ,(car xy)] [,py ,(Multiset M) ,(cadr xy)]})
9                (match-all t (List M)
10                 [(join hs (cons x ts)) `(,x ,(append hs ts))]))]
11         ...))))
```

Here is another example from the `List` matcher. The cost of pattern matching for the `join` pattern is high. However, the cost for enumerating only the tail parts is low; we can write a tail recursive function for enumerating the tail parts. When we use join patterns, the first argument of `join` is often a wildcard. Therefore, by adding a match clause for a `join` pattern whose first argument is a wildcard as follows, we can improve the efficiency of pattern matching for a list.

```
1  (define List
2    (lambda (M)
3      (lambda (p t)
4        (match p
5          [('nil) (if (eq? t '{}) '{{}} '{})]
6          [('cons px py)
7           (match t
8                  (() '{})
9                  ((x . xs)
10                  `{{[,px ,M ,x] [,py ,(List M) ,xs]}}))]
11         [('join '_ py)
12          (map (lambda (y) `{[,py ,(List M) ,y]})
13               (tails t))]
14         [('join px py)
15          (map (lambda (xy) `{[,px ,(List M) ,(car xy)] [,py ,(List M) ,(cadr xy)]})
16               (unjoin t))]
```

```
        [('val x) (if (eq? x t) '{{}} '{})]
        [pvar `{{[,pvar Something ,t]}}])))
```

6.2 Benchmark Result

I compared the performance of a nested cons pattern for a multiset. For benchmarking Egison and the proposed Scheme macros, I used the expressions below. between is a function of Egison that returns a list that contains sequential integers between the first and second argument.

```
(match-all (between 1 n) (multiset something) [<cons $x <cons $y _>> [x y]])
```

Similarly, iota is a function in Scheme that returns a list of sequential integers that starts from the integer of the second argument.

```
(match-all (iota n 1) (Multiset Something) [(cons x (cons y _)) `(,x ,y)])
```

I also made a comparison with the functional description of the same program in Scheme.

```
(define comb2 (lambda (xs) (comb2-helper xs '{})))

(define comb2-helper
  (lambda (xs hs)
    (if (eq? xs '{})
        '{}
        (append (append (map (lambda (y) `(,(car xs) ,y)) hs)
                        (map (lambda (y) `(,(car xs) ,y)) (cdr xs)))
                (comb2-helper (cdr xs) (append hs `(,(car xs))))))))

(comb2 (iota n 1))
```

I used Egison version 3.8.1 and Gauche version 0.9.7 on 2.3 GHz Intel Core i5 processor for this benchmark.

comb2	n=50	n = 100	n = 200	n=400	n=800	n=1600
Egison with multiset in Sect. 5.1	1.189s	4.470s	21.441s	112.67s	697.66s	n/a
Egison with multiset in Sect. 6.1	0.438s	1.312s	4.751s	22.612s	112.89s	714.43s
Scheme with Multiset in Sect. 5.1	0.124s	0.436s	2.413s	17.900s	117.41s	n/a
Scheme with Multiset in Sect. 6.1	0.074s	0.099s	0.259s	0.752s	3.159s	12.641s
Scheme (functional style)	0.026s	0.042s	0.107s	0.373s	1.618s	7.949s

Table 1. Benchmark results

Table 1 shows the benchmark results. I ran all the benchmarks five times and took a median for each of them. For all the benchmarks, Scheme is faster than Egison. The reason for these performance differences is that Gauche, a scheme implementation I used for the benchmark, has a compiler while Egison has only a simple interpreter implementation.

Adding a matcher clause for an inductive pattern with wildcards is effective for both Egison and Scheme. This trick improves the performance of Egison and Scheme around 6 times and 40 times respectively for $n = 800$. The performance improvement is larger for Scheme than Egison because Scheme is a strict language. Scheme always evaluates "(append hs ts)" in the 9th line of the Multiset definition in Sect. 5.1 whereas Egison evaluates the corresponding expression only when it is necessary.

We can also observe that pattern-matching-oriented programming style is only two times slower than functional programming style in this case, though the program in pattern-matching-oriented programming style much simpler.

7 FUTURE WORK

Implementation of more expressive pattern constructs for non-free data types remains as future work. For example, several advanced pattern constructs in Egison such as *loop patterns* [19], *sequential patterns*, and *pattern functions* [6] are still not implemented in the proposed Scheme macros. Loop patterns allow us to describe repetitions in a pattern like the Kleene star operator of the regular expressions. Loop patterns are more expressive than the other repeated pattern such as the Kleene star operator in the sense that loop patterns allow users to change the pattern repeated depending on the current repeat count [19]. Sequential patterns are generalization of later patterns and allow users to control the order of the pattern-matching process. Pattern functions allow users to modularize patterns with lexical scope by composing patterns. These pattern constructs widen the range of patterns that we can describe concisely. Currently, I am preparing a paper to show usefulness of these patterns.

Integration of Egison pattern matching with a programming language with a static type system also remains as future work. Though we have an experimental Template Haskell implementation [7], this Haskell implementation currently has a limitation that it does not allow users to define new pattern constructors. For example, the `nil`, `cons`, and `join` pattern constructors are built-in in the current implementation. This limitation comes from the limitation of Haskell that does not allow overlapping data families. [5] In the current Template Haskell implementation of Egison, patterns are defined in the library as data of "`Pattern a`", which is a data type for patterns against data whose type is "`a`". Users cannot add new data constructors in "`Pattern a`" in their program for adding new pattern constructors because of overlap is not permitted. Currently, we are working to find a method for avoiding this problem.

8 CONCLUSION

This paper proposed a method for compiling Egison pattern-matching expressions that can describe very expressive pattern matching for non-free data types but has a complicated pattern-matching procedure. The proposed method made an implementation of the pattern-matching system of Egison very short; the macros and the pattern-matching procedure is implemented in less than 150 lines of Scheme code. This implementation is also a good proof of extensibility of Scheme and Lisp.

This pattern matching library will help the research of pattern matching by providing an easy method for developing new pattern-matching extensions. This pattern matching library will also help the education of algorithms by making its implementation simpler as the implementation of a SAT solver presented in this paper. I hope this work leads to the further development and propagation of advanced pattern matching.

ACKNOWLEDGMENTS

I thank Yuichi Nishiwaki for constructive discussion while implementing the proposed Scheme macros and writing this paper. I thank Tatsuya Yamashita for the first implementation of a SAT solver in Egison. I thank Yuito Murase for implementing the proposed macros in Common Lisp. I thank Mayuko Kori for implementing the proposed macros using Template Haskell. Finally, I greatly thank my shepherd William Byrd and the anonymous reviewers of the 20th Scheme and Functional Programming Workshop for their reviews that dramatically improved this paper.

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. https://github.com/egison/egison-scheme. [Online; accessed 21-Feb-2019].
[3] 2018. Gauche – A Scheme Implementation. http://practical-scheme.net/gauche. [Online; accessed 1-June-2019].
[4] 2018. zeptometer/egison-common-lisp: Common Lisp Macros for Egison Pattern Matching. https://github.com/zeptometer/egison-common-lisp. [Online; accessed 4-June-2019].
[5] 2019. 13.1. Language options — Glasgow Haskell Compiler 8.6.5 User's Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html. [Online; accessed 4-June-2019].
[6] 2019. Egison Manual - Basics of Patterns. https://www.egison.org/manual/patterns.html. [Online; accessed 4-June-2019].
[7] 2019. egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. https://github.com/egison/egison-haskell. [Online; accessed 21-Feb-2019].
[8] 2019. Gauche Users' Reference: Pattern matching. https://practical-scheme.net/gauche/man/gauche-refe/Pattern-matching.html. [Online; accessed 4-June-2019].
[9] 2019. Gauche Users' Reference: Prime numbers. https://practical-scheme.net/gauche/man/gauche-refe/Prime-numbers.html. [Online; accessed 5-July-2019].
[10] 2019. Gauche Users' Reference: Stream library. https://practical-scheme.net/gauche/man/gauche-refe/Stream-library.html. [Online; accessed 5-July-2019].
[11] 2019. Gauche Users' Reference: Traditional macros. https://practical-scheme.net/gauche/man/gauche-refe/Traditional-macros.html. [Online; accessed 4-June-2019].
[12] 2019. Guile Reference Manual: Pattern Matching. https://www.gnu.org/software/guile/docs/docs-2.0/guile-ref/Pattern-Matching.html. [Online; accessed 4-June-2019].
[13] 2019. Pattern matching - The CHICKEN Scheme wiki. https://wiki.call-cc.org/man/3/Pattern-matching. [Online; accessed 4-June-2019].
[14] Sergio Antoy. 2001. Constructor-based conditional narrowing. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*.
[15] Sergio Antoy. 2010. Programming with narrowing: A tutorial. *Journal of Symbolic Computation* 45, 5 (2010).
[16] Philip L Bewig. 2007. Scheme Request for Implementation 41: Streams. (2007).
[17] Rod M. Burstall. 1969. Proving properties of programs by structural induction. *Comput. J.* 12, 1 (1969), 41–48.
[18] 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.
[19] 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*.
[20] 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.
[21] Martin Erwig. 1996. Active patterns. *Implementation of Functional Languages* (1996).
[22] Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.
[23] Michael Hanus. 2007. Multi-paradigm declarative languages. In *International Conference on Logic Programming*.
[24] John Harrison. 2009. *Handbook of practical logic and automated reasoning.* Cambridge University Press.
[25] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy With Class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*. ACM Press, 1–55.
[26] F. V. McBride, D. J. T. Morrison, and R. M. Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969).
[27] Simon Peyton Jones. 1987. *The implementation of functional programming languages.* Prentice-Hall, Inc.
[28] Christian Queinnec. 1990. Compilation of non-linear, second order patterns on S-expressions. In *Programming Language Implementation and Logic Programming*.
[29] Alex Shinn, John Cowan, Arthur A Gleckler, et al. 2013. Revised 7 Report on the Algorithmic Language Scheme.
[30] Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised 6 report on the algorithmic language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301.
[31] Simon Thompson. 1986. Laws in Miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming*.
[32] Simon Thompson. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming* 13, 2-3 (1990).
[33] Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).
[34] Mark Tullsen. 2000. First Class Patterns. *Practical Aspects of Declarative Languages*.
[35] David Turner. 2012. Some history of functional programming languages. In *International Symposium on Trends in Functional Programming*. Springer, 1–20.
[36] 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*.
[37] Andrew K. Wright and Bruce F. Duba. 1993. Pattern matching for Scheme. *Unpublished manuscript* (1993).

A PATTERN-MATCHING-ORIENTED PROGRAMMING

The pattern-matching facility of Egison is so expressive that a new programming style called *pattern-matching-oriented programming style* arises. This programming style makes descriptions of mathematical algorithms concise by confining explicit recursions for traversing data inside intuitive patterns. This section shows several sample programs in this programming style. I am currently preparing another paper for discussing pattern-matching-oriented programming techniques. We can execute programs shown in this section by loading a program in [2] on Gauche [3].

A.1 Basic List Processing Functions in Pattern-Matching-Oriented Style

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

The `map` function is defined using pattern matching as follows. The `(join _ (cons x _))` pattern matches each element of a target list. We call this pattern *join-cons pattern* because it often appears in list programming. Combining a join-cons pattern with `match-all`, we can simply implement the `map` function.

```
(define pm-map
  (lambda (f xs)
    (match-all xs (List Something)
      [(join _ (cons x _)) (f x)])))

(pm-map (lambda (x) (+ x 10)) `(1 2 3 4))
; (11 12 13 14)
```

By doubly nesting the above join-cons pattern, we can define the `concat` function. Note that, we can create a matcher for such as a list of lists and a multiset of multisets by composing matchers.

```
(define pm-concat
  (lambda (xss)
    (match-all xss (List (List Something))
      [(join _ (cons (join _ (cons x _)) _)) x])))

(pm-concat `((1 2) (3) (4 5)))
; (1 2 3 4 5)
```

By combining a not-pattern with a doubly-nested join-cons pattern, we can also define a `unique` function. The pattern below extracts only the last appearance of each element; a not-pattern is used to describe that there is no more `x` after an occurrence of `x`. `Eq` is a matcher that can handle pattern matching for a value pattern. `eq?` is used for checking equality.

```
(define pm-unique-simple
  (lambda (xs)
    (match-all xs (List Eq)
      [(join _ (cons x (not (join _ (cons ,x _)))) x])))

(pm-unique-simple `(1 2 3 2 4))
; (1 3 2 4)
```

We can define `unique` whose results consist of the first appearance of each element by using a later pattern. A later pattern is used to describe that there is no appearance of `x` in the target list for the first argument of the `join` pattern.

```
(define pm-unique
  (lambda (xs)
    (match-all xs (List Eq)
      [(join (later (not (join _ (cons ,x _)))) (cons x _)) x])))

(pm-unique `(1 2 3 2 4))
; (1 2 3 4)
```

A.2 Implementation of a SAT Solver

The program below describes the Davis-Putnam algorithm. We can see a full implementation of this SAT solver in `dp.scm` in the GitHub repository of the proposed Scheme macros [2]. Pattern matching for multisets dramatically improves the readability of the description of this algorithm. We can compare this Scheme program with the OCaml implementation of the same algorithm in [24].

The `sat` function takes two argument `vars`, a list of propositional variable, and `cnf`, a propositional logic formula in conjunctive normal form. The tuple consists of `vars` and `cnf` is pattern-matched as a tuple of a multiset of integers and a multiset of multisets of integers in the definition of `sat`. In this program, a propositional variable is represented as a positive integer, and a literal is represented as a integer. For example, -1 represents negation of the propositional variable 1.

```
(define sat
  (lambda [vars cnf]
    (match-first `[,vars ,cnf] `[,(Multiset Integer) ,(Multiset (Multiset Integer))]
      ; satisfiable
      ['[_ ()] #t]
      ; unsatisfiable
      ['[_ (cons () _)] #f]
      ; 1-literal rule
      ['[_ (cons (cons l ()) _)]
       (sat (delete (abs l) vars) (assign-true l cnf))]
      ; pure literal rule (positive)
      ['[(cons v vs) (not (cons (cons ,(neg v) _) _))]
       (sat vs (assign-true v cnf))]
      ; pure literal rule (negative)
      ['[(cons v vs) (not (cons (cons ,v _) _))]
       (sat vs (assign-true (neg v) cnf))]
      ; otherwise
      ['[(cons v vs) _]
       (sat vs (append (resolve-on v cnf)
               (delete-clauses-with v (delete-clauses-with (neg v) cnf))))])))
```

The first match clause (line 5) matches if `cnf` is empty. This match clause returns `#t` because `cnf` is satisfiable in this case. The second match clause (line 7) matches if `cnf` contains an empty clause. This match clause returns `#f` because `cnf` is unsatisfiable in this case. The third match clause (lines 9 and 10) matches if `cnf` contains a clause that consists of a single literal `l`. In this match clause, we assign `l` true at once. The fourth match clause (lines 12 and 13) matches when there is a propositional variable `v` that appears only positively in `cnf`. In this match clause, we assign `v` true at once. The fifth match clause (lines 15 and 16) matches when there is a propositional variable `v` that appears only negatively in `cnf`. In this match clause, we assign `v` false at once. The final match clause

(lines 18-20) matches when pattern matching for all the above match clauses fails. This match clause applies the resolution principle.

B PATTERN MATCHING WITH INFINITELY MANY RESULTS

`match-all` provided by `stream-egison.scm` supports pattern matching with infinitely many results. A library for streams described in SRFI 41 [16] is used for handling lazy lists. The following `match-all` expressions extract an infinite list of twin primes and prime triplets from a stream of prime numbers.

```
(load "./stream-egison.scm")

(define stream-primes (stream-filter bpsw-prime? (stream-iota -1 1)))

(stream->list
 (stream-take
  (match-all stream-primes (List Integer)
             [(join _ (cons p (cons ,(+ p 2) _)))
              `(,p ,(+ p 2))])
  10))
; ((3 5) (5 7) (11 13) (17 19) (29 31) (41 43) (59 61) (71 73) (101 103) (107 109))

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

`bpsw-prime?` is a predicate that checks whether the argument positive integer is a prime number or not. This predicate is provided by the `math.prime` Gauche library [9]. `stream-iota`, `stream->list`, and `stream-take` are provided by the `util.stream` Gauche library [10]. "`(stream-iota -1 1)`" returns a stream that contains all the positive integers. `stream-take` returns a stream that contains the first elements of the first argument stream. The number of elements taken from the given stream is specified by the second argument. `stream->list` is a function that converts the given stream to a list.

```
(stream->list (stream-take (stream-iota -1 1) 10))
; (1 2 3 4 5 6 7 8 9 10)
```

In the languages whose default evaluation strategy is non-strict as Haskell and Egison, we do not need to distinguish these two `match-all` implementations.