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, 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), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 modern functional programming languages such as Haskell and OCaml support pattern matching for algebraic data types. Many Scheme implementations also have a similar pattern-matching facility [6] though Scheme does not contain pattern matching for algebraic data types in its specification, R7RS [20].

However, there are data types that are not algebraic data types and for which we cannt use the common pattern-matching facility. 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 and implemented for extending the range of data types to which pattern matching is applicable [17, 25]. Among them, a pattern-matching system proposed by Egi and Nishiwaki [14] 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)

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

XXXX-XXXX/2019/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 they 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 for importing Egison pattern matching. These macros have been already open-sourced in [2]. We can try them on the Gauche Scheme interpreter [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 usage of the proposed Scheme macros for pattern matching. Sect. 4 shows practical examples of pattern matching for non-free data types. Sect. 5 explains the pattern matching algorithm of Egison and implementation of this algorithm in Scheme. Sect. 6 explains the implementation of the proposed macros. Sect. 7 mentions the remained work. Finally, Sect. 8 concludes the paper.

2 RELATED WORK

This section review a 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 [11]. Burstall proposed to use the notation "`let cons(a, y) = x`" instead of "`let (a, y) = decons(x)`". In [11], `concat` is defined in the modern fashion.

After that, user-defined algebraic data types are invented. HOPE [12] 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 1980s, more expressive pattern matching for wider range of data types started to be pursued. Miranda's laws [21, 22] and Wadler's views [26] 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 [26] 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 [15] proposed in 1996 and Tullsen's first class patterns [24] proposed in 2000 are such extensions. Both extensions allow users to customize the pattern-matching algorithm for each pattern constructor. 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 [14]. The expressions below match a collection that consists of n zeros as a multiset of integers with patterns that match a collection with sequential pairs and triples. The `match-all` expression used in the program below returns a collection evaluating the body expression for all the pattern=matching results. The target collection contains neither sequential pairs nor triples. As a result, both expression return an empty collection. Egison uses backtracking for traversing a search tree. Therefore, the time for returning the result is same for both expressions.

```
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])
```

```
; returns [] in O(n^2) time
```

The program below defines a pattern-matching algorithm for a multiset. The `matcher` expression used in the following program is a built-in syntax of Egison for defining pattern-matching algorithms for each data type.

```
(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}]}]})))
```

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 [18], which was developed on top of Lisp. This pattern-matching system supports non-linear patterns. 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.

Queinnec [19] also pursued more expressive pattern matching. Though this proposal is specialized to lists and not extensible, the proposed language supports the `cons` and the `join` patterns, `match-all`, not-patterns, and recursive patterns. His proposal achieves almost perfect expressiveness for patterns against lists.

We can regard Egison as an extension of them that adds the customizability of pattern-matching algorithms.

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 [27] is implemented in most of the well-known Scheme implementations such as Gauche [6], Guile [7], and Chicken [8].

Among them, Racket [9] provides more expressive pattern matching than other Scheme implementations. The `match` expander [23] of Racket allows users arbitrary transformation of data when pattern matching. For example, we can describe view patterns [26] using the `match` expander.

This paper implement 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 BASIC USAGE OF PROPOSED MACROS

This section explains the usage of the Scheme library for Egison pattern matching.

3.1 Syntax

Here, I show the formal grammar of pattern-matching expressions of the proposed Scheme macros.

⟨*pm-expr*⟩ ::= '`(match-all` ⟨*expr*⟩ ⟨*expr*⟩ ⟨*match-clause*⟩ (match-all)
| '`(match-first` ⟨*expr*⟩ ⟨*expr*⟩ ⟨*match-clause*⟩* (match)

⟨*match-clause*⟩ ::= '`[`'⟨*pat*⟩ ⟨*expr*⟩'`]`' (match clause)

⟨*pattern*⟩ ::= '`_`' (wildcard)
| ⟨*ident*⟩ (pattern variable)
| '`,`' '`(`' ⟨*pattern*⟩* '`)`' (tuple pattern)
| '`(`' ⟨*ident*⟩ ⟨*pattern*⟩* '`)`' (inductive pattern)
| '`,`'⟨*expr*⟩ (value pattern)
| '`(or`' ⟨*pattern*⟩* '`)`' (or-pattern)
| '`(and`' ⟨*pattern*⟩* '`)`' (and-pattern)
| '`(not`' ⟨*pattern*⟩ '`)`' (not-pattern)
| '`(later`' ⟨*pattern*⟩ '`)`' (later pattern)

The `match-all` expression has the completely same meaning with that of Egison. The `match-first` has similar to Wright's `match` expression and it evaluate the body of the first match clause where the pattern matches with the target. The `match-first` expression can take multiple match clause.

This pattern-matching library provides several pattern constructs. The following subsection explains each of them.

3.2 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.

3.3 Tuple Patterns

Tuple patterns are represented by prepending "`,`" to a list of patterns. Each element of a tuple pattern is pattern-matched with a corresponding target using a corresponding matcher. (TODO: fix)

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

3.4 Inductive Patterns

When the pattern is a list, the first element is always handed 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.

3.5 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.

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.

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

3.6 Logical Patterns

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

```
(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.

```
(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.

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

3.7 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.

```
(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, this section demonstrate the pattern matching facility implemented in this paper to show its usefulness. 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 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 doubling 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, we can define a `unique` function. A not-pattern is used to describe that there is no more `x` after an occurrence of `x`. Therefore, this pattern extracts only the last appearance of each element.

```
(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.

```
(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)
```

4.2 Implementation of SAT Solver

The program below describes the Davis-Putnam algorithm. We can see a full implementation of this SAT solver in `dp.scm` of [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 [16].

```
(define sat
  (lambda [vars cnf]
    (match-first `[,vars ,cnf] `[,(Multiset Integer) ,(Multiset (Multiset Integer))]
      ['[_ ()] #t]
      ['[_ (cons () _)] #f]
      ['[_ (cons (cons l ()) _)]
       (sat (delete (abs l) vars) (assign-true l cnf))]
      ['[(cons v vs) (not (cons (cons ,(neg v) _) _))]
       (sat vs (assign-true v cnf))]
      ['[(cons v vs) (not (cons (cons ,v _) _))]
       (sat vs (assign-true (neg v) cnf))]
      ['[(cons v vs) _]
       (sat vs (append (resolve-on v cnf)
              (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. A library for streams described in SRFI 41 [10] 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)))

```

```
(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))
```

5 ALGORITHM OF EGISON PATTERN MATCHING

This section explains and implements the pattern-matching algorithm of Egison, which is described in Sect. 5 and Sect. 7 of [14]. After the explanation of the pattern-matching algorithm, I introduce a simplified implementation of this algorithm in Scheme. This implementation is called by the macros whose implementation are introduced in Sect. 6.

In Egison, pattern matching is implemented as reductions of stacks of *matching atoms*. A matching atom is a triple of a pattern, target, and matcher. In each step of a pattern-matching process, the top matching atom is popped out. From this matching atom, a list of lists of next matching atoms is calculated. Each list of the next matching atoms is pushed to the stack of matching atoms. As a result, a single stack of matching atom is reduced to multiple stacks of matching atoms in a reduction step. Pattern matching is recursively executed for each stack of matching atoms. When a stack becomes empty, it means pattern matching for this stack succeeded. Sect. 5.1 of [14] shows a sample of this reduction.

The `processMState` function below implements this process. `processMState` takes a *matching state* as its argument. A matching atom consists of a stack of matching atoms and an intermediate result of pattern matching.

The match clause in the 4th and 5th lines describes the case where the matcher of the top matching atom is `Something`. In this case, the value `t` is added to the intermediate pattern-matching result. `Something` is the only built-in matcher of the Egison pattern-matching system. `something` can handle only wildcards or pattern variables, and is a only matcher that can bind a value to a pattern variable. The reason why an intermediate result is a list of values and not a list of pairs of a symbol and a value is explained in Sect. 6.2.

The match clause in the 6th-8th lines describes the general case. The list of lists of next matching atoms are calculated in the 7th 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.

In the program below, 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 here. We can see the full implementation of `processMState` in `egison.scm` of [2].

```
(define processMState
  (lambda (mState)
    (match mState
      ...
      (('MState {[pvar 'Something t] . mStack} ret)
```

```
6       `((MState ,mStack ,(append ret `(,t)))))
7      (('MState {[p M t] . mStack} ret)
8       (let {[next-matomss (M p t)]}
9         (map (lambda (next-matoms) `(MState ,(append next-matoms mStack) ,ret)) next-matomss))))))
```

The processMStates below implements the whole pattern-matching process while the above processMState implements a step of this pattern-matching process. processMStates takes a list of matching states as its argument and returns all pattern-matching results. The first match clause (the 5th line) describes the case where a list of matching states is empty. In this case, processMStates simply returns an empty list and terminates. The second match clause (the 6th and 7th line) 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 rest matching states. The third match clause (the 8th and 9th line) 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 rest matching states.

```
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))))))
```

Here, I simplify the algorithm by omitting the function to traverse an infinitely large search tree. I omit the explanation of the algorithm for enumerating infinite results of pattern matching in this paper. It is implemented in stream-egison.scm of [2].

6 IMPLEMENTATION METHOD

This section introduces a method for implementing the match-all macro in Scheme. Three tricks, which are explained in Sect. 6.1, Sect. 6.2, and Sect. 6.3 respectively, enable this implementation.

6.1 Compiling Matchers to lambdas

In the original paper of Egison pattern matching [14], a matcher is a special object that retains a pattern-matching algorithm. In the proposed Scheme macros, for encoding a matcher with lambda to avoid introducing new built-in objects, a matcher is defined 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.

```
1 (define Multiset
2   (lambda (M)
3     (lambda (p t)
4       (match p
5         (('nil) (if (eq? t '()) '{{}} '{}))
6         (('cons px py)
7          (map (lambda (xy) `{[,px ,M ,(car xy)] [,py ,(Multiset M) ,(cadr xy)]})
8               (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]}})))))
```

In the above program, we uses "`{}`" and "`[]`" separately. We use braces "`{}`" for enclosing a list. For example, a list of matching atoms is enclosed with braces. We use brackets "`[]`" for enclosing a tuple. For example, a matching atom, which is a triple of a pattern, a matcher, and a target, is enclosed with brackets.

As a matcher definition of Egison [14] that consists of matcher clauses that describe pattern matching for patterns to describe pattern-matching algorithms for each pattern, the above matcher definition describe pattern matching for patterns. For pattern matching against patterns, `match` is used.

The first match clause (the 5th line) describes a 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 (the 6th-9th lines) 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 (the 10th-14th line) 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 (the 15th line) 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`.

6.2 Compiling `match-all` to 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. The `gen-match-results` function 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 above transformation is done by The following macro.

```
1 (define-macro (match-all t M clause)
2   (let* {[p (rewrite-pattern (list 'quasiquote (car clause)))]
```

```
        [e (cadr clause)]}
   `(map (lambda (ret) (apply (lambda ,(extract-pattern-variables p) ,e) ret))
        (gen-match-results ,p ,M ,t))))
```

The program below defines `gen-match-results`. `gen-match-results` is a macro that just calls the `processMStates` function introduced in Sect. 5. An initial matching state "`(MState {[p M t]} {})`" is created from "`p`", "`M`", and "`t`".

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

Thus, by separating the extraction of pattern variables and the calculation of pattern-matching results, we can compile such a complicated pattern matching process. However, this separation disable to apply the existing method for compiling pattern matching efficiently. (TODO: FIX)

6.3 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. 5.

```
(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 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. 6.1, for example.

7 FUTURE WORK

7.1 Importation of More Expressive Pattern Constructs

Extending our macros for supporting loop patterns [13] is also future work.

7.2 Integration with Statically Typed Languages

Integration of Egison pattern matching with a programming language with a static type system remains as future work. We have an experimental Template Haskell implementation [5]. Currently, this Haskell implementation has a problem that it does not allow users to define new pattern constructors. The Haskell program below shows the definition of `Pattern a`, which is a data type for patterns against data whose type is `a`. Haskell does not allow to add new data constructors to a data type defined in a different file. Therefore, users cannot add a new pattern constructor in their program.

```
data Pattern a where
  Wildcard :: Pattern a
```

```
3   PatVar :: String -> Pattern a
4   ValuePat :: Eq a => ([Result] -> a) -> Pattern a
5   ValuePat' :: Eq a => a -> Pattern a
6   AndPat :: Pattern a -> Pattern a -> Pattern a
7   OrPat :: Pattern a -> Pattern a -> Pattern a
8   NotPat :: Pattern a -> Pattern a
9   LaterPat :: Pattern a -> Pattern a
10  PredicatePat :: (a -> Bool) -> Pattern a
11  NilPat :: a ~ [b] => Pattern a
12  ConsPat :: a ~ [b] => Pattern b -> Pattern a -> Pattern a
13  JoinPat :: a ~ [b] => Pattern a -> Pattern a -> Pattern a
```

8 CONCLUSION

This paper proposed a method for compiling a complicate pattern-matching procedure for functional programming languages with high-ordered functions. extraction of pattern variables and destruction of targets. The proposed method allows us to implement very expressive pattern matching in a short program; the macros and the pattern-matching procedure is implemented in less than 150 lines of code (`egison.scm` of [2]). This is a good proof of extensibility of Scheme.(TODO: fix)

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

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] 2018. zeptometer/egison-common-lisp: Common Lisp Macros for Egison Pattern Matching. [Online; accessed 4-June-2019].
[5] 2019. egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. [Online; accessed 21-Feb-2019].
[6] 2019. Gauche Users' Reference: Pattern matching. https://practical-scheme.net/gauche/man/gauche-refe/Pattern-matching.html. [Online; accessed 4-June-2019].
[7] 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].
[8] 2019. Pattern matching - The CHICKEN Scheme wiki. https://wiki.call-cc.org/man/3/Pattern-matching. [Online; accessed 4-June-2019].
[9] 2019. Racket. https://racket-lang.org/. [Online; accessed 4-June-2019].
[10] Philip L Bewig. 2007. Scheme Request for Implementation 41: Streams. (2007).
[11] Rod M. Burstall. 1969. Proving properties of programs by structural induction. *Comput. J.* 12, 1 (1969), 41–48.
[12] 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.
[13] 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*.
[14] 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.
[15] Martin Erwig. 1996. Active patterns. *Implementation of Functional Languages* (1996).
[16] John Harrison. 2009. *Handbook of practical logic and automated reasoning*. Cambridge University Press.
[17] 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.
[18] F. V. McBride, D. J. T. Morrison, and R. M. Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969).
[19] Christian Queinnec. 1990. Compilation of non-linear, second order patterns on S-expressions. In *Programming Language Implementation and Logic Programming*.

[20] Alex Shinn, John Cowan, Arthur A Gleckler, et al. 2013. Revised 7 Report on the Algorithmic Language Scheme.
[21] Simon Thompson. 1986. Laws in miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming*.
[22] Simon Thompson. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming* 13, 2-3 (1990).
[23] Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).
[24] Mark Tullsen. 2000. First Class Patterns. *Practical Aspects of Declarative Languages*.
[25] David Turner. 2012. Some history of functional programming languages. In *International Symposium on Trends in Functional Programming*. Springer, 1–20.
[26] 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*.
[27] A.K. Wright and B.F. Duba. 1993. Pattern matching for Scheme. *Unpublished manuscript* (1993).