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. The key idea of this method is 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.(TODO: more detail) 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

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 [7] though Scheme does not contain pattern matching for algebraic data types in its specification, R7RS [22].

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 and implemented for extending the range of data types to which pattern matching is applicable [19, 27]. Among them, a pattern-matching system proposed by Egi and Nishiwaki [15] 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

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

(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 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 explains the pattern matching algorithm of Egison and implementation 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 [12]. Burstall proposed to use the notation "let cons(a, y) = x" instead of "let (a, y) = decons(x)". In [12], concat is defined in the modern fashion. Algebraic data types were introduced after pattern matching. HOPE [13] 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 [23, 24] and Wadler's views [28] 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 on views [28] present 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 more expressive extensions of pattern matching. Erwig's active patterns [16] proposed in 1996 and Tullsen's first class patterns [26] 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 [15]. The expressions below match a list "{1 2 3}" as a list of integers "(list integer)" using the pattern "<cons $x $ts>".

```
1 (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. 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. In Egison, a pattern

variable begin with "$". 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 do pattern matching 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 as follows.

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

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

The expressions below match a list that consists of n zeros "(take n (repeat 0))" as a multiset of integers "(multiset integer)" 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(TODO: cite). Therefore, the time complexity for the same pattern matching is $O(n^3)$ in Curry. Backtracking is important feature for handling non-linear patterns.

```
1 (match-all (take n (repeat 0)) (multiset integer) [<cons $x <cons ,(+ x 1) <cons ,(+ x 2) _>>> x])
2 ; 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. Egison allows users to define their own matchers.

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. The definition of multiset in Scheme will be explained in Sect. 5.1. I will skip the explanation of the following program for now.

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

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 symbolic mathematical expression was implemented in the symbol manipulation system proposed by McBride [20], 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 [21] 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 [29] is implemented in most of the well-known Scheme implementations such as Gauche [7], Guile [9], and Chicken [10].

Among them, Racket [17] provides more expressive pattern matching than other Scheme implementations. The match expander [25] of Racket allows users arbitrary transformation of data when pattern matching. For example, we can describe view patterns [28] 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 USAGE OF PROPOSED MACROS

This section introduces the usage of the pattern-matching expressions provided by the proposed Scheme macros.

3.1 Syntax Design

Here is the formal grammar of the pattern-matching expressions of the proposed Scheme macros. M denotes an expression. p denotes a pattern. x denotes a symbol. C denotes a pattern constructor.

$$M ::= (\text{match-all } M \ M \ [p \ M] \ \cdots)$$
$$| \ (\text{match-first } M \ M \ [p \ M] \ \cdots)$$
$$| \ \text{Something}$$

$$p ::= \_ \ | \ x \ | \ ,M \ | \ (C \ p \ \cdots) \ | \ '(p \ \cdots)$$
$$| \ (\text{or } p \ \cdots) \ | \ (\text{and } p \ \cdots) \ | \ (\text{not } p)$$
$$| \ (\text{later } p)$$

(TODO: segmentation variables and loop patterns)
We 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. Though we use <> for applying a pattern constructor in Egison, we use () for that in Scheme because we cannot use <> as parenthesis in Scheme. (TODO: refer to R6RS and R7RS for the specification of parenthesis.)

The `match-all` expression has the completely same meaning as that of Egison. The `match-first` has 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`. 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.

This pattern-matching library provides several pattern constructs. The following subsections explain each of them.

3.1.1 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.

3.1.2 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.

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

3.1.3 Tuple Patterns. (TODO: The reason of the syntax design)
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.

```
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))
```

3.1.4 Value Patterns. (TODO: The reason of the syntax design)
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 3 2 1) as a multiset. The pattern in the program below matches if the target collection contains pairs of elements in a sequence.

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

3.1.5 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.

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

3.1.6 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 succeeded.

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

3.2 Pattern-Matching-Oriented Programming

The expressive power of Egison pattern matching 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.(TODO: fix) 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].

3.2.1 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 [18].

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.

```
1  (define sat
2    (lambda [vars cnf]
3      (match-first `[,vars ,cnf] `[,(Multiset Integer) ,(Multiset (Multiset Integer))]
4        ; satisfiable
5        ['[_ ()] #t]
6        ; unsatisfiable
7        ['[_ (cons () _)] #f]
8        ; 1-literal rule
9        ['[_ (cons (cons l ()) _)]
10        (sat (delete (abs l) vars) (assign-true l cnf))]
11        ; pure literal rule (positive)
12        ['[(cons v vs) (not (cons (cons ,(neg v) _) _))]
13        (sat vs (assign-true v cnf))]
14        ; pure literal rule (negative)
15        ['[(cons v vs) (not (cons (cons ,v _) _))]
16        (sat vs (assign-true (neg v) cnf))]
17        ; otherwise
18        ['[(cons v vs) _]
19        (sat vs (append (resolve-on v cnf)
20                (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.

3.2.2 Redefinitions of Basic List Processing Functions. (TODO: more detail explanation) 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.

```
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)
```

We can define unique whose results consist of the first appearance of each element by using a later pattern.

```
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 ALGORITHM OF EGISON PATTERN MATCHING

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 [15]. 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 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 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. Sect. 5.1 of [15] shows a sample of this reduction.

The processMState function below implements this process. processMState takes a matching state as its argument and returns a list of next matching states. In the program below, 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. `Something` is the only built-in matcher of the Egison pattern-matching system. `Something` can handle only wildcards or pattern variables, and is an 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 value is explained in Sect. 5.2.

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.

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. We can see the full implementation of `processMState` in `egison.scm` of [2].

```
1 (define processMState
2   (lambda (mState)
3     (match mState
4       ...
5       (('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 (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 rest 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 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))))))
```

The above implementation is from `egison.scm` of [2] that does not support pattern matching with infinitely many results. In `processMStates` of `stream-egison.scm`, the traversal of an infinitely large search tree is implemented that is explained in Sect. 5.2 of the original Egison paper [15]. Appendix A demonstrates pattern matching that uses the macros provided by `stream-egison.scm`.

(TODO: Implementation of `match-first`) This distinct implementation of `processMStates` is not necessary for the Egison language that adopt a lazy evaluation strategy.

```
1 (define processMStates1
2   (lambda (mStates)
3     (match mStates
4          (() '())
5          ((('MState '{} ret) . rs)
6           `(,ret))
7          ((mState . rs)
8           (processMStates1 (append (processMState mState) rs)))
9          )))
```

5 IMPLEMENTATION METHOD

This section introduces a method for implementing the `match-all` macro in Scheme. The three tricks that are explained in Sect. 5.1, Sect. 5.2, and Sect. 5.3 enable this implementation.

5.1 Compiling Matchers to `lambda`s

In the Egison pattern matching system [15], 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 to enhance the readability of the program. 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 [15], 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 [29] 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 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 '(join hs (cons x ts)))`" returns "`(hs x ts)`". 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 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 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))))
```

`define-macro` in the above program is a traditional non-hygienic macro similar to `defmacro` of Common Lisp [8]. 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 …(TODO: find a good verb) in R7RS(TODO: cite) is difficult. By using `syntax-case` and `syntax->datum` in R6RS, we can define the above macro relatively easily.

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

```
1 (define-macro (gen-match-results p M t)
2   `(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 the large pattern matching procedure of Egison.

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.

```
1 (define processMState
2   (lambda (mState)
3     (match mState
4       (('MState {[('val f) M t] . mStack} ret)
5        (let {[next-matomss (M `(val ,(apply f ret)) t)]}
6          (map (lambda (next-matoms) `(MState ,(append next-matoms mStack) ,ret)) next-matomss)))
7       ...)))
```

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 collection. The cost for calculating the rest 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         (('nil) (if (eq? t '()) '{{}} '{}))
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          (('val v)
12           (match-first `(,t ,v) `(,(List M) ,(Multiset M))
13             ('((nil) (nil)) '{{}})
14             ('((cons x xs) (cons ,x ,xs)) '{{}})
15             ('(_ _) '{})))
16          (pvar `{{[,pvar Something ,t]}})))))
```

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)))
17         (('val x) (if (eq? x t) '{{}} '{}))
18         (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 following expressions.

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

```
1 (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.

```
1 (define comb2 (lambda (xs) (comb2-helper xs '())))
2
3 (define comb2-helper
4   (lambda (xs hs)
5     (if (eq? xs '())
6         '()
7         (append (append (map (lambda (y) `(,(car xs) ,y)) hs)
```

```
8                    (map (lambda (y) `(,(car xs) ,y)) (cdr xs)))
9              (comb2-helper (cdr xs) (append hs `(,(car xs))))))))))
10
11 (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	1.189	4.470	21.441	112.67	697.66	n/a
Egison with `multiset` in Sect. 6.1	0.438	1.312	4.751	22.612	112.89	714.43
Scheme with `Multiset` in Sect. 5.1	0.124	0.436	2.413	17.900	117.41	n/a
Scheme with `Multiset` in Sect. 6.1	0.074	0.099	0.259	0.752	3.159	12.641
Scheme (functional style)	0.026	0.042	0.107	0.373	1.618	7.949

For all the benchmark, 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

(TODO: write something)

7.1 Importation of More Expressive Pattern Constructs

Several advanced pattern constructs for non-free data types such as loop patterns [14], sequential patterns, and pattern functions [5] are still not implemented in the proposed Scheme macros. Implementation of these pattern constructs is 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. Though we have an experimental Template Haskell implementation [6], this Haskell implementation currently 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.(TODO: cite: data family overlapping) Therefore, users cannot add a new pattern constructor in their program. Currently, we are working to find a method for avoiding this problem. (TODO: explanation of the following code)

```
1 data Pattern a where
2   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 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.

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. [Online; accessed 4-June-2019].
[5] 2019. Egison Manual - Basics of Patterns. [Online; accessed 4-June-2019].
[6] 2019. egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. https://github.com/egison/egison-haskell. [Online; accessed 21-Feb-2019].
[7] 2019. Gauche Users' Reference: Pattern matching. https://practical-scheme.net/gauche/man/gauche-refe/Pattern-matching.html. [Online; accessed 4-June-2019].
[8] 2019. Gauche Users' Reference: Traditional macros. https://practical-scheme.net/gauche/man/gauche-refe/Traditional-macros.html. [Online; accessed 4-June-2019].
[9] 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].
[10] 2019. Pattern matching - The CHICKEN Scheme wiki. https://wiki.call-cc.org/man/3/Pattern-matching. [Online; accessed 4-June-2019].
[11] Philip L Bewig. 2007. Scheme Request for Implementation 41: Streams. (2007).
[12] Rod M. Burstall. 1969. Proving properties of programs by structural induction. *Comput. J.* 12, 1 (1969), 41–48.
[13] 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.
[14] 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*.
[15] 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.
[16] Martin Erwig. 1996. Active patterns. *Implementation of Functional Languages* (1996).
[17] Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.
[18] John Harrison. 2009. *Handbook of practical logic and automated reasoning.* Cambridge University Press.
[19] 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.
[20] F. V. McBride, D. J. T. Morrison, and R. M. Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969).

[21] Christian Queinnec. 1990. Compilation of non-linear, second order patterns on S-expressions. In *Programming Language Implementation and Logic Programming*.

[22] Alex Shinn, John Cowan, Arthur A Gleckler, et al. 2013. Revised 7 Report on the Algorithmic Language Scheme.

[23] Simon Thompson. 1986. Laws in Miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming*.

[24] Simon Thompson. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming* 13, 2-3 (1990).

[25] Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).

[26] Mark Tullsen. 2000. First Class Patterns. *Practical Aspects of Declarative Languages*.

[27] David Turner. 2012. Some history of functional programming languages. In *International Symposium on Trends in Functional Programming*. Springer, 1–20.

[28] 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*.

[29] Andrew K. Wright and Bruce F. Duba. 1993. Pattern matching for Scheme. *Unpublished manuscript* (1993).

A 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 [11] 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.

(TODO: bpsw-prime?, stream-iota, stream-take)

```
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))
```

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.

B 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. In this paper, 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.

```
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)
```

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.

```
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)
```