Egison Tutorial: Functional Programming in Pattern-Matching-Oriented Programming Style

Satoshi Egi, Yuichi Nishiwaki

May 30, 2019

Abstract

Throughout the history of functional programming, recursion has been emanating as a natural method for describing loops in programs. However, there does often exist a substantial cognitive distance between the recursive definition and the simplest explanation of an algorithm even for the basic list processing functions such as `map`, `concat`, or `unique`; when we explain these functions, we seldom use recursion explicitly as we do in functional programming. For example, `map` is often explained as follows: the map function takes a function and a list and returns a list of the results of applying the function to all the elements of the list.

This paper introduces a new programming style called *pattern-matching-oriented* programming for filling this gap. An essential ingredient of our method is utilizing pattern matching for non-free data types. Pattern matching for non-free data types features non-linear pattern matching with backtracking and extensibility of pattern-matching algorithms. Several non-standard pattern constructs such as not-patterns, loop patterns, and sequential patterns are derived from this pattern-matching facility. Based on that result, this paper introduces many programming techniques that replace explicit recursions with an intuitive pattern by confining recursions inside a pattern. We classify these techniques as pattern-matching-oriented programming design patterns.

These programming techniques allow us to redefine not only the most basic functions for list processing such as `map`, `concat`, or `unique` more elegantly than the traditional functional programming style, but also more practical mathematical algorithms and software such as a SAT solver, computer algebra system, and database query language that we had not been able to implement concisely.

1 Introduction

How do you answer the question "What is the map function?" We believe most people answer as follows:

(1) "The map function takes a function and a list and returns a list of the results of applying the function to all the elements of the list."

Few people answer as follows:

(2) "The map function takes a function and a list and returns an empty list if the argument list is empty. Otherwise, it returns a list whose head element is the result of applying the function to the head element of the argument list, and the tail part is the result of applying the map function recursively to the tail part of the argument list."

Obviously, there is a significant gap between these two explanations. The former explanation is simpler and more straightforward than the latter. However, the current functional definition of `map` is based on the latter as follows.

```
map _ [] = []
map f (x : xs) = (f x) : (map f xs)
```

Interestingly, this definition has been almost unchanged for 60 years since McCarthy first presented the definition of `maplist` in [17]. The only difference is a way for describing conditional branches: McCarthy uses predicates, whereas Haskell uses pattern matching.

```
maplist[x ;f] = [null[x] -> NIL; T -> cons[f[x]; maplist[cdr[x]; f]]]
```

Recursion used in the above definition is a mathematically simple but powerful framework for representing computations and has been a very basic construct of functional programming. Recursion is heavily used for definitions of many basic functions such as `filter`, `concat`, and `unique` and most of them are also simple enough. Throughout the history of functional programming, recursion emanates as the natural method for describing loops in programs. For example, recursion schemes [18] are famous work that classifies programming patterns in functional programming.

However, as mentioned earlier, there *does* exist a substantial cognitive distance between the definition and the simplest explanation of that definition. To illustrate this gap, we define two variations of `map`.

The first one is `adjacentMap`. `adjacentMap` takes a function of two arguments and a list, and returns a list of applying the function for all the adjacent pairs in the list. The conditional branch for the recursive definition of `adjacentMap` is not as simple as that of `map`.

```
adjacentMap _ [] = []
adjacentMap _ [_] = []
adjacentMap f [x, y] = [f x y]
adjacentMap f (x : y : xs) = (f x y) : (adjacentMap f (y : xs))
```

The second one is `mapWithBothSides`. `mapWithBothSides` takes a function of three arguments and a list, and returns a list of applying the function for all three-tuples consisting of the head, the current element, and the tail. `mapWithBothSides` is used for generating lists by rewriting the element of an input list. This function is useful for handling logical formulae, for example. The definition of `mapWithBothSides` gets more complicated than `adjacentMap`. We define a helper function for `mapWithBothSides` as follows.

```
mapWithBothSides f xs = mapWithBothSides' f [] xs
 where
  mapWithBothSides' f xs [] = []
  mapWithBothSides' f xs (y : ys) = (f xs y ys) : (mapWithBothSides' f (xs ++ [y]) ys)
```

The explanations for these variations of `map` is similar to (1), and their definitions should be similar to each other. However, their definitions are very different from each other. A hint for filling the gap is hidden behind these differences.

The cause of these difference is lack of a pattern like "`hs ++ ts`" that divides a target list into head and tail parts. We call this pattern a *join pattern*. Unlike traditional pattern matching for algebraic data types, this pattern has multiple decompositions. Pattern matching for *non-free data types* is necessary for handling join patterns.

A data type is called non-free if there are multiple different forms for equivalent data. For example, multisets, sets, graphs, and polynomials are non-free data types. Lists with join constructors are also non-free data types because the list `[1,2]` has multiple forms with `Join`: `Join [] [1,2]`, `Join [1] [2]`, and `Join [1,2] []`, for example.

In Egison [23] whose distinguishing feature is a pattern matching for non-free data types, we can define `map`, `adjacentMap`, and `mapWithBothSides` concisely and in the very similar way as follows.[1][2]

```
(define $map
  (lambda [$f $xs]
    (match-all xs (list something)
      [<join _ <cons $x _>> (f x)])
```

```
(define $adjacent-map
  (lambda [$f $xs]
    (match-all xs (list something)
      [<join _ <cons $x <cons $y _>>> (f x y)])
```

```
(define $map-with-both-sides
  (lambda [$f $xs]
    (match-all xs (list something)
      [<join $hs <cons $x $ts>> (f hs x ts)])
```

[1]In this paper, we present Egison in Haskell-like syntax to omit the explanation of the syntax detail, though Egison originally has Lisp-like syntax.

[2]The creator of Egison, an author of this paper, got an idea of the language when he implemented `mapWithBothSides` for implementing an automated theorem conjecturer.

The above definitions are close to the explanation of `map` given in (1). We achieved this by hiding the recursion in the definition of `list`, which defines the pattern-matching algorithm for join patterns. We call this programming style *pattern-matching-oriented programming style*. The above examples show just a part of expressiveness of pattern-matching-oriented programming. This paper introduces full features of the Egison pattern-matching-oriented programming language and presents all the techniques we discovered so far for replacing explicit recursions with an intuitive pattern.

The remainder of this paper is organized as follows. Sect. 2 reviews the history of pattern matching and sees how pattern matching has evolved to extend its target data types to non-free data types. Sect. 3 introduces Egison and various pattern constructs for non-free data types. These pattern constructs increase the number of situations in which we can replace verbose recursions with more intuitive patterns. Sect. 4 catalogs pattern-matching-oriented programming techniques utilizing the features introduced in Sect. 3. Sect. 5 explores the effect of pattern-matching-oriented programming in more practical situations. Finally, Sect. 6 concludes the paper.

2 Related Work: Evolution of Pattern Matching

Abstractions of algorithms consist of two kinds of abstractions: procedural abstraction and data abstraction. Pattern matching is an important feature of data abstraction. Before the invention of pattern matching, code for decomposing data makes programs verbose. For example, we had to write many `car` and `cdr` to decompose a list. Pattern matching relieved programmers of this burden by allowing them to describe data decomposition only with an intuitive pattern. When pattern matching first appeared, pattern matching could only be applied to the specific types of algebraic data types. Huge efforts have been conducted to remove this limitation. Pattern matching has evolved by bringing the programs that were written in the body of match clauses into the definitions of patterns. This importation has gradually proceeded. As a result, state-of-the-art work allows us pattern matching for non-free data types. This section reviews this evolution by seeing what happened in each decade.

The early history of pattern matching is described in detail also in a paper on the history of Haskell [15] and Turner's paper [29].

2.1 1960s-1970s: Dawn of Pattern Matching

Burstall's paper in 1969 [4] proposed pattern matching that looks similar to the pattern-matching facility that is widely used nowadays. Burstall proposed to use the notation "`let cons(a, y) = x`" instead of "`let (a, y) = decons(x)`".[3] The following program is the first functional program that uses pattern matching.

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

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 [16], which was developed on top of Lisp. This pattern-matching system supports *non-linear patterns*. A non-linear pattern is a pattern that allows multiple occurrences of the same variables in a pattern. Their paper demonstrates some examples that process symbolic mathematical expressions to show the expressive power of non-linear patterns. However, this approach does not support pattern matching with multiple results, and users cannot extend its pattern-matching facility.

User-defined algebraic data types are invented in 1970s. HOPE [5] by Burstall, MacQueen, and Sannella is a well-known language that introduced user-defined algebraic data types and pattern matching for them. The following is the definition of binary trees presented in this paper.

```
data tree(alpha) == empty ++ tip(alpha) ++ node(tree(alpha)#tree(alpha))
```

[3]The latter notation had already been supported by ISWIM that supports pattern matching for tuples.

2.2 1980s: Spread of Pattern Matching and Invention of Views

In this decade, functional languages with user-defined algebraic data types and pattern matching for them became common. Miranda by Turner [28] and Haskell [15] were the most popular languages among these languages, and the first pattern-matching extensions for widening the target of pattern matching beyond algebraic data types were designed on them.

Miranda's laws [26, 25] and Wadler's views [30] 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 [30] 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.

At the same time, more expressive pattern matching is explored by Queinnec [20], who proposed an expressive pattern matching for lists. Though this proposal is specialized to lists and not extensible, the proposed language supports the `cons` and the `join` patterns, non-linear pattern matching with backtracking, `matchAll`, not-patterns, and recursive patterns. His proposal achieves almost perfect expressiveness for patterns of lists and allows the pattern-matching-oriented definition of the basic list processing functions. For example, the following `member` definition is presented in Queinnec's paper [20].

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

2.3 1990s and 2000s: Exploration for Expressive Patterns

Following the pattern-matching extensions in the previous decade, several new pattern-matching extensions for extending the target range of pattern matching have been proposed by several researchers. We review these proposals in this section.

2.3.1 Erwig's Active Patterns

Erwig's active patterns [11] are an attempt to extend the expressiveness of patterns beyond Wadler's views. Active patterns also allow users to customize the pattern-matching algorithm for each pattern. `Add'` in the following program is a pattern constructor of active patterns. `Add'` extracts an element that is identical with the first argument of `Add'` from the target collection.

```
pat Add' (x,_) =
  Add (y,s) => if x == y then Add (y,s)
                         else let Add' (x,t) = s
                                in Add (x, Add (y, t)) end
```

Using the above `Add'`, we can define the `member` function as follows, hiding the recursion as the pattern-matching-oriented definition of `map` we presented in Sect. 1.

```
fun member x (Add' (x,s)) = true
  | member x s            = false
```

However, the expressiveness of active patterns is still limited. Active patterns do not support pattern matching with multiple results: `Add'` can take only a value and cannot take a pattern variable as its first argument. Non-linear patterns exhibit their full ability when they are combined with pattern matching with backtracking.

2.3.2 Tullsen's First Class Patterns

Tullsen's first class patterns [27] are another extension of views. First class patterns provide a sophisticated syntax for defining user-defined patterns. It allows users to directly define a method for decomposing data for each pattern constructor. For example, the pattern constructor "`cons#`" that decomposes a list in the join representation is defined as follows. The syntax of *next-target expressions* of Egison, which will be explained in Appendix A, is similar to that of first class patterns.

```
data List a = Nil | Unit a | Join (List a) (List a)

cons# Nil = Nothing
cons# (Unit a) = Just (a,Nil)
cons# (Join xs ys) = case cons# xs of
                       Just (x,xs') -> Just (x, Join xs' ys)
                       Nothing      -> cons# ys
```

First class patterns support pattern matching with multiple results. We can define pattern constructors that have multiple decompositions simply by changing the type of the return value of pattern constructors from `Maybe` to `List`. However, the expressiveness of first class patterns is still limited because it does not support non-linear patterns. Non-linear pattern matching is a necessary feature for describing useful patterns for non-free data types.

2.3.3 Functional Logic Programming

Functional logic programming is an independent approach for pattern matching against non-free data types. Curry [13] by Hanus is the most popular functional logic programming language. Curry supports both properties: non-linear patterns and pattern matching with multiple results. Therefore, we can write expressive patterns for non-free data types in Curry.

However, Curry does not intend to utilize non-linear pattern matching with multiple results fully. For example, non-linear pattern matching is not efficiently executed in Curry. Theoretical time complexities depend on the size of patterns as follows.[4] The reason for the difference of time complexities between these patterns is that Curry transforms non-linear patterns into pattern guards [2, 1, 12]. Pattern guards are applied after enumerating all pattern-matching results. Therefore, substantial unnecessary enumerations often occur before the application of pattern guards.

```
seq2 (insert x (insert (x+1) _)) = "Matched"
seq2 _ = "Not matched"

seq3 (insert x (insert (x+1) (insert (x+2) _))) = "Matched"
seq3 _ = "Not matched"

seq2 (take 10 (repeat 0)) -- returns "Not matched" in O(n^2) time
seq3 (take 10 (repeat 0)) -- returns "Not matched" in O(n^3) time
```

Moreover, Curry does not provide a special syntax construct for handling multiple pattern-matching results. Curry provides `findall` for handling multiple unification results. However, if we use `findall` for pattern matching, the program gets more complicated than the functional approach. For example, Curry program that defines the `map` function in the pattern-matching-oriented style is as follows.

```
map f xs = findall (\y -> let x free in (_ ++ (x : _)) =:= xs & f x =:= y)
```

2.4 2010s: Toward a Unified Theory of Pattern-Matching-Oriented Programming

In this decade, a unified theory for practical pattern matching for non-free data types has been pursued. Egison [9] proposed by the same authors of this paper is such research. The research listed and organized the properties for practical pattern matching for non-free data types. They proposed three criteria in their paper. The criteria are as follows:

1. Efficiency of the backtracking algorithm for non-linear patterns,
2. Extensibility of pattern matching, and
3. Polymorphism in patterns.

[4][9] shows benchmark results of `seq2` and `seq3`.

The first two properties are the properties that had already been discussed in the previous decades, though they had not been discussed simultaneously. Egison fulfills both properties as follows.

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

The above expressions match a collection that consists of *n* zeros as a multiset of integers for enumerating sequential pairs and triples, respectively. This target collection contains neither sequential pairs nor triplets, therefore both expressions return an empty collection.

`matchAll` is a built-in syntax construct of Egison that collects all the pattern-matching results and returns a collection where the body expression has been evaluated for each result. `matchAll` takes one argument *matcher* that is "`multiset integer`" in the above case. A matcher is an Egison specific object that knows how to decompose the target following the given pattern. The matcher is specified between "`as`" and "`with`", which are reserved words. "`multiset`" is a user-defined function that takes a matcher for the elements and returns a matcher for multisets. "`multiset`" defines a method for interpreting the cons (`:`) pattern. "`integer`" is a user-defined matcher that is used for pattern-matching an integer. "`_`" that appears in a pattern is a wildcard. Pattern variables are prepended with "$". An expression in a pattern following "#" is called a *value pattern*. A value pattern is evaluated and compared with the target. In the following example, "`#x`" is a value pattern. As a result, "`multiset integer`" is evaluated as a matcher for pattern-matching a multiset of integers.

The pattern-matching algorithm inside Egison includes the backtracking mechanism for efficient non-linear pattern matching. In the above case, Egison interpreter does not try pattern matching for "`#(x + 2)`" because pattern matching for "`#(x + 1)`" always fails. Therefore, the time complexities of the above expressions are identical. For reading this paper, we do not need to know this pattern-matching algorithm, which is discussed in [9] in detail.

Moreover, Egison uses call-by-need evaluation, and `matchAll` is evaluated lazily. Therefore, only the necessary matches are calculated at the pattern-matching process.

The third is a new property that was discussed for the first time in Egison. Polymorphic patterns are important for non-free data types because some data are pattern-matched as various non-free data types at the different parts of a program. For example, a collection is pattern-matched as a list, a multiset, and a set. Polymorphic patterns reduce the number of names for pattern constructors.

In the following sample, a list "`[1,2,3]`" is pattern-matched using different matchers with the same cons pattern. In the case of sets, the rest elements are the same as the original collection because we ignore the redundant elements. If we interpret a set as a collection that contains infinitely many copies of each element, this specification of cons for sets is natural.

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

Polymorphic patterns are useful especially when we use value patterns. As well as other patterns, the behavior of value patterns is dependent on matchers. For example, an equality "`[1,2,3] == [2,1,3]`" between collections is false if we regard them as mere lists but true if we regard them as multisets. Still, thanks to polymorphism of patterns, we can use the same syntax for both types. This dramatically improves the readability of the program and makes programming with non-free data types easy.

```
(match-all {1 2 3} (list integer) [,{2 1 3} "Matched"]) ; {}
(match-all {1 2 3} (multiset integer) [,{2 1 3} "Matched"]) ; {"Matched"}
```

The rest of this paper discusses programming techniques utilizing Egison.

3 Quick Tour of the Egison Pattern-Matching-Oriented Language

This section introduces the built-in syntax constructs for pattern matching in Egison. Most of them are discussed also in [9] and [7]. Sequential patterns in Sect 3.5 are newly proposed in this paper.

3.1 `matchAll` and `matchAllDFS` for Handling Multiple Pattern-Matching Results

The `matchAll` expression is designed to enumerate all countably infinite pattern-matching results. For this purpose, users sometimes need to care about the order of pattern-matching results.

Let us start by showing a representative sample. The `matchAll` expression below enumerates all pairs of natural numbers. We extract the first 8 elements with the `take` function. `matchAll` traverses the reduction tree of pattern matching in breadth-first search to traverse all the nodes (Sect. 5.2 of [9]). As a result, the order of the pattern-matching results is as follows.

```
(take 8 (match-all nats (set something) [<cons $x <cons $y _>> [x y]]))
; {[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3]}
```

The above order is preferable for traversing an infinitely large reduction tree. However, sometimes, this order is not preferable (see Sect. 4.1.2 and 4.4.1). `matchAllDFS` that traverses a reduction tree in depth-first order is provided for this reason.

```
(take 8 (match-all-dfs nats (set something) [<cons $x <cons $y _>> [x y]]))
; {[1 1] [1 2] [1 3] [1 4] [1 5] [1 6] [1 7] [1 8]}
```

In the above sample, the `something` matcher is used. `something` is a matcher that can be used for arbitrary objects but can handle only pattern variables and wildcards. `something` is the only built-in matcher in Egison.

3.2 Value Patterns and Predicate Patterns for Representing Non-linear Patterns

`matchAll` gets even more powerful when combined with non-linear patterns. For example, the following non-linear pattern matches when the target collection contains two identical elements.

```
(match-all {1 2 3 2 4 3} (list integer) [<join _ <cons $x <join _ <cons ,x _>>>> x]) ; {2 3}
```

Value patterns play an important role in representing non-linear patterns. A value pattern matches the target if the target is equal to the content of the value pattern. A value pattern is prepended with "#" and the expression after "#" is evaluated referring to the value bound to the pattern variables that appear on the left side of the patterns. As a result, for example, "`$x : #x : _`" is valid, but "`#x : $x : _`" is invalid.

Let us show pattern matching for *twin primes* as a sample of non-linear patterns. Twin primes are pairs of prime numbers whose forms are $(p, p+2)$. `primes` is an infinite list of prime numbers. This `matchAll` extracts all twin primes from this infinite list of prime numbers.

```
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ,(+ p 2) _>>> [p (+ p 2)]]))

(take 8 twin-primes) ; {[3 5] [5 7] [11 13] [17 19] [29 31] [41 43] [59 61] [71 73]}
```

There are cases that we might want to use more general predicates in patterns than equality. Predicate patterns are provided for such a purpose. A predicate pattern matches the target if the predicate returns true for the target. A predicate pattern is prepended with "?", and a predicate of one argument follows after "?".

```
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ?(lambda [$q] (eq? q (+ p 2))) _>>> [p (+ p 2)]]))
```

3.3 Logical Pattern Constructs: And-Patterns, Or-Patterns, and Not-Patterns

The situations where and-patterns and or-patterns are useful are similar to those of the existing languages, whereas not-patterns become useful when they are combined with non-linear pattern matching with backtracking.

We start by showing pattern matching for *prime triplets* as an example of and-patterns and or-patterns. A prime triplet is a triplet of primes whose form is $(p, p+2, p+6)$ or $(p, p+4, p+6)$. The and-pattern is used as an as-pattern. The or-pattern is used to match both of $p+2$ and $p+4$.

```
(define $prime-triplets
  (match-all primes (list integer)
    [<join _ <cons $p <cons (& (| ,(+ p 2) ,(+ p 4)) $m) <cons ,(+ p 6) _>>>> [p m (+ p 6)]]))

(take 8 prime-triplets)
; {[5 7 11] [7 11 13] [11 13 17] [13 17 19] [17 19 23] [37 41 43] [41 43 47] [67 71 73]}
```

A not-pattern matches a target if the pattern does not match the target, as its name implies. A not-pattern is prepended with "!", and a pattern follows after "!". The following `matchAll` enumerates sequential pairs of prime numbers that are *not* twin primes.

```
(take 10 (match-all primes (list integer)
           [<join _ <cons $p (& !#(p + 2) $q) _>> [p q]]))
; {[2 3] [7 11] [13 17] [19 23] [23 29] [31 37] [37 41] [43 47] [47 53] [53 59]}
```

3.4 Loop Patterns for Representing Repetition

A loop pattern is a pattern construct for representing a pattern that repeats multiple times. It is an extension of Kleene star operator of regular expressions for general non-free data types [7].

Let us start by considering pattern matching for enumerating all combinations of two elements from a target collection. It can be written using `matchAll` as follows.

```
(define $comb2
  (lambda [$xs]
    (match-all xs (list integer)
      [<join _ <cons $x_1 <join _ <cons $x_2 _>>>>
       {x_1 x_2}])))

(comb2 {1 2 3 4}) ; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
```

Egison allows users to append indices to a pattern variable as `$x_1` and `$x_2` in the above sample. They are called *indexed variables* and represent x_1 and x_2 in mathematical expressions. The expression after "`_`" must be evaluated to an integer and is called an *index*. We can append as many indices as we want like "`x_i_j_k`". When a value is bound to an indexed pattern variable `$x_i`, the system initiates an abstract map consisting of key-value pairs if `x` is not bound to a map, and bind it to `x`. If `x` is already bound to a map, a new key-value pair is added to this map.

Now, we generalize `comb2`. The loop patterns can be used for that purpose.

```
(define $comb
  (lambda [$n $xs]
    (match-all xs (list integer)
      [(loop $i [1 {n} _]
         <join _ <cons $x_i ...>>
         _)
       (map (lambda [$i] x_i) (between 1 n))])))

(comb 2 {1 2 3 4}) ; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
(comb 3 {1 2 3 4}) ; {{1 2 3} {1 2 4} {1 3 4} {2 3 4}}
```

The loop pattern takes an *index variable*, *index range*, *repeat pattern*, and *end pattern* as arguments. An index variable is a variable to hold the current repeat count. An index range specifies the range where the index variable moves. A repeat pattern is a pattern repeated when the index variable is in the index range. An end pattern is a pattern expanded when the index variable gets out of the index range.

Inside loop patterns, we can use the *ellipsis pattern* (`...`). The repeat pattern or the end pattern is expanded at the location of the ellipsis pattern. The repeat pattern is expanded replacing the ellipsis pattern incrementing the value of the index variable.

The repeat counts of the loop patterns in the above samples are constants. However, we can also write a loop pattern whose repeat count varies depending on the target by specifying a pattern instead of an integer as the end number. When an end number is a pattern, the ellipsis pattern is replaced with both the repeat pattern and the end pattern, and the repeat count when the ellipsis pattern is replaced with the end pattern is pattern-matched with that pattern. The following loop pattern enumerates all head parts of the target collection.

```
(match-all {1 2 3 4} (list something)
  [(loop $i [1 $n] <cons $x_i  ...> _)
   (map (lambda [$i] x_i) (between 1 n))])
; {{} {1} {1 2} {1 2 3} {1 2 3 4}}
```

Loop patterns are heavily used especially for trees and graphs. We work on pattern matching for trees in Sect. 4.4.1.

3.5 Sequential Patterns for Controlling the Order of Pattern-Matching Process

The pattern-matching system of Egison processes patterns from left to right in order. However, there are cases where we want to change this order, for example, to refer to the value bound to the right side of a pattern. Sequential patterns are provided for such a purpose.

Sequential patterns allow users to control the order of the pattern-matching process. A sequential pattern is represented as a list of patterns. Pattern matching is executed for each pattern in order. In the following sample, the target list is pattern-matched from the third, first, and second element in order.

```
(match-all {2 3 1 4 5} (list integer)
  [{<cons # <cons # <cons $x _>>>
    [,(+ x 1) #]
    ,(+ x 2)}
   "Matched"])
; {"Matched"}
```

"@" that appears in a sequential pattern is called *later pattern variable*. The target data bound to later pattern variables are pattern-matched in the next sequence. When multiple later pattern variables appear, they are pattern-matched as a tuple in the next sequence. It allows us to apply not-patterns for different parts of a pattern at the same time as we will see in Sect 4.3.

Some readers might wonder that a sequential pattern can be transformed into a nested `match-all` expression. There are at least two reasons why it is impossible. First, a nested `match-all` expression breaks breadth-first search strategy: the inner `match-all` for the second result of the outer `match-all` is executed only after the inner `match-all` for the first result of the outer `match-all` is finished. Second, a later pattern variable retains the information of not only a target but also a matcher. There are cases that the matcher of `match-all` is a parameter passed as an argument of a function, and a pattern is polymorphic. Therefore, it is impossible to determine the matchers of inner `match-all` expressions syntactically.

3.6 Matcher Compositions

Matchers are composable, and we can define matchers for such as tuples of multisets and multisets of multisets. Using this feature, we can define matchers for various data types.

First, we can define a matcher for tuples by a tuple of matchers. A tuple pattern is used for pattern matching using such a matcher. For example, we can define the `intersect` function using a matcher for tuples of two multisets. We work on pattern matching for tuples of collections more in Sect. 4.3.

```
(define $intersect
  (lambda [$xs $ys]
    (match-all [xs ys] [(multiset something) (multiset eq)]
      [[<cons $x _> <cons ,x _>] x])))
```

eq is a user-defined matcher for data types for which equality is defined. When the eq matcher is used, equality is checked for a value pattern.[5]

By passing a tuple matcher to a function that takes and returns a matcher, we can define a matcher for various non-free data types. For example, we can define a matcher for a graph as a set of edges. In the following code, we assume a node id is represented by an integer.

[5]A definition of the eq matcher is explained in Sect. 6.3 of [9].

```
(define $graph (multiset [integer integer]))
```

A matcher for adjacency graphs also can be defined. An adjacency graph is defined as a multiset of tuples of an integer and a multiset of integers.

```
(define $adjacency-graph (multiset [integer (multiset integer)]))
```

Some readers might wonder about matchers for algebraic data types. Egison provides a special syntax construct for defining a matcher for an algebraic data type. For example, a matcher for binary trees can be defined using `algebraicDataMatcher`.

```
(define $binary-tree
  (lambda [$a]
    (algebraic-data-matcher
      {<b-leaf a> <b-node a (binary-tree a) (binary-tree a)>})))
```

Matchers for algebraic data types and matchers for non-free data types also can be combined. For example, we can define a matcher for trees whose nodes have an arbitrary number of children whose order is ignorable. We show pattern matching for these trees in Sect. 4.4.1.

```
(define $tree
  (lambda [$a]
    (algebraic-data-matcher
      {<leaf a> <node a (multiset (tree a))>})))
```

4 Pattern-Matching-Oriented Programming Design Patterns

This section introduces basic pattern-matching-oriented programming techniques that replace explicit recursions to intuitive patterns. In the first part of this section, we rewrite many list processing functions such as `map`, `filter`, `elem`, `delete`, `any`, `every`, `unique`, `concat`, and `difference`, for which we expect most functional programmers imagine the same definitions. In the latter part of this section, we move our focus to descriptions of more mathematical algorithms that are not well supported in the current functional programming languages. We proceed with this section by listing patterns that frequently appear and show situations in which they are useful. The following table shows this list.

Name	Description	Explained in	Used in
Join-cons pattern for list	Enumerate combinations of elements.	4.1	
Cons pattern for multiset	Enumerate permutations of elements.	4.2	4.3, 4.4, 5.1
Tuple pattern for collections	Compare multiple collections.	4.3	4.4, 5.1
Loop pattern	Describe repetitions inside patterns.	4.4	

4.1 Join-Cons Patterns for Lists — List Basic Functions

Join patterns whose second argument is a cons pattern, such as "`_ ++ $x : _`", are frequently used for lists. We call these patterns *join-cons patterns*. Many basic list processing functions can be redefined by simply using this pattern.

4.1.1 Single Join-Cons Patterns — The `map` Function and Its Family

"`_ ++ $x : _`" matches each element of the target collection when the `list` matcher is used. As a result, the `matchAll` expression below matches each element of `xs`, and returns the results of applying `f` to each of them. As discussed in Introduction, this `map` definition is very close to the explanation of `map` ((1) of Sect 1).

```
(define $map
  (lambda [$f $xs]
    (match-all xs (list something)
      [<join _ <cons $x _>> (f x)])))
```

By modifying the above `matchAll` expression, we can define several functions. For example, we can define `filter` by inserting a predicate pattern.

```
(define $filter
  (lambda [$p $xs]
    (match-all xs (list something)
      [<join _ <cons (& ?p $x) _>> x])))
```

We can define `elem` by using a value pattern. `elem` is a predicate that determines whether the first argument element appears in the second argument list or not. `match` is provided also in Egison. `match` is just an alias of "`car (matchAll ...)`" because Egison evaluates `matchAll` lazily.[6]

```
(define $member?
  (lambda [$x $xs]
    (match xs (list eq)
      {[<join _ <cons ,x _>> #t]
       [_ #f]})))
```

We can define `delete` that remove the first appearance of `x` from `xs` by modifying `elem`.

```
(define $delete
  (lambda [$x $xs]
    (match xs (list eq)
      {[<join $hs <cons ,x $ts>> {@hs @ts}]
       [_ #f]})))
```

The predicate `any` and `every` [22] also can be concisely defined with predicate patterns using `match`. `any` is a predicate that determines whether any element of the second argument list satisfies the first argument predicate. `every` is a predicate that determines whether all elements of the second argument list satisfy the first argument predicate.

```
(define $any
  (lambda [$p $xs]
    (match xs (list something)
      {[<join _ <cons ?p _>> #t]
       [_ #f]})))

(define $every
  (lambda [$p $xs]
    (match xs (list something)
      {[<join _ <cons !?p _>> #f]
       [_ #t]})))
```

4.1.2 Nested Join-Cons Patterns — The `unique` and `concat` Function

By combining multiple join-cons patterns, we can describe more expressive patterns. One example is the `unique` function. The `unique` function is defined in the pattern-matching-oriented style as follows.

```
(define $unique (lambda [$xs] (match-all xs (list eq) [<join _ <cons $x !<join _ <cons ,x _>>>> x])))
```

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.

```
(unique {1 2 3 2 4}) ; {1 3 2 4}
```

We can define `unique` whose results consist of the first appearance of each element by rewriting the above pattern using a predicate pattern with the `elem` predicate. To match only the first appearance of an element, we rewrite a pattern that ensures that the same element does not appear before that element. We cannot write such a pattern with a simple combination of the cons and join patterns because they match a target list from left to right.

[6] `matchAll` also can handle multiple match clauses. "`matchAll` t `as` m `with` c_1 c_2 …" is equivalent to "`matchAll` t `as` m `with` c_1 `++` `matchAll` t `as` m `with` c_2 `++` …".

```
(define $unique
  (lambda [$xs]
    (match-all xs (list eq)
      [<join $hs <cons (& !?(lambda [$x] (member? x hs)) $x) _>> x])))

(unique {1 2 3 2 4}) ; {1 2 3 4}
```

Another more elegant solution is using a sequential pattern. We can describe the same pattern by using the sequential pattern for the first argument of join.

```
(define $unique
  (lambda [$xs]
    (match-all xs (list eq)
      [{<join # <cons $x _>> <join _ <cons ,x _>>} x])))
```

Another example of a nested join-cons pattern is `concat`. We can define `concat` in the pattern-matching-oriented style by combining a nested join-cons pattern and matcher composition. Note that `matchAllDFS` is necessary for ordering the output list properly.

```
(define $concat
  (lambda [$xss]
    (match-all-dfs xss (list (list something))
      [<join _ <join _  <cons $x _>> _> x])))
```

If we used `matchAll` instead of `matchAllDFS` for `concat`, it enumerates the elements of the input list of lists in circler order.

```
(match-all {nats (map negate nats)} (list (list something))
  [<join _ <join _  <cons $x _>> _> x]
; {1 2 -1 3 -2 4 -3 5 -4 6}
```

4.2 Cons Patterns for Multisets

A single cons pattern for a multiset can replace a join-cons pattern for a list. For example, `elem` can be defined using a cons pattern for multiset as follows.

```
(define $member?
  (lambda [$x $xs]
    (match xs (multiset eq)
      {[<cons ,x _> #t]
       [_ #f]})))
```

The `lookup` function for association lists can also be defined similarly.

```
(define $lookup
  (lambda [$x $ls]
    (match ls (multiset [eq something])
      {[<cons [,k $x] _> x]})))
```

The usage of cons patterns for multisets differs from that of join-cons patterns when they are nested. Cons patterns for multisets can be used to enumerate $P(n,k)$ permutations of k elements, whereas join-cons patterns can be used to enumerate $C(n,k)$ combinations of k elements.

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

The descriptions of algorithms for which nested cons patterns are suitable become complicated in the traditional functional style. We can see that by just comparing the descriptions of the above two `matchAll` in functional programming.

However, pattern matching for multisets often appears in mathematical algorithms. Besides that, a much wider variety of patterns exist for multisets than lists. As a result, functions that correspond to patterns for multisets are not implemented as library functions because naming all these patterns is not practical. In functional programming so far, they are defined as a recursive function or combining several functions by users each time. It makes functional descriptions of mathematical algorithms complicated.

Thus, descriptions of these mathematical algorithms are the area where pattern-matching-oriented programming demonstrates its full power. The rest of this paper discusses how we can describe these wide variety of patterns for multisets by just combining pattern constructs introduced in Sect. 3.

4.3 Tuple Patterns with Sequential Not-Patterns for Comparing Collections

When describing algorithms, we often meet a situation to compare multiple data. A tuple pattern combined with not-patterns is especially useful for this purpose. For example, we can define `difference` by inserting a not-pattern into the definition of `intersect` in Sect. 3.6.

```
(define $difference
  (lambda [$xs $ys]
    (match-all [xs ys] [(multiset eq) (multiset eq)]
      [[<cons $x _> !<cons ,x _>] x])))
```

By changing the position of the not-pattern as "`!($x : _, #x : _)`", we can also describe a pattern that matches when no common element exists between two collections.

We can write more complicated patterns by combining these patterns with a sequential pattern that allows us to apply a not-pattern to separate parts of the pattern simultaneously. For example, a pattern that matches when only one common element exists between the two collections is described below. A sequential pattern enables us to describe the pattern-matching process that first extracts one common element from the two collections, and after that checks that no common element exists between the rest two collections. Sequential not-patterns often appear in mathematical algorithms, and we show an example again in Sect. 5.1.

```
(define $only-one-common-element
  (lambda [$xs $ys]
    (match-all [xs ys] [(multiset eq) (multiset eq)]
      {[{[<cons $x #> !<cons ,x #>]
         ![<cons $y _> !<cons ,y _>]}
        #t]
       [_ #f]})))
```

We can combine a sequential pattern also with a loop pattern. For example, we can write a pattern that matches the common head elements of two lists with a sequential loop pattern.

```
(define $common-heads
  (lambda [$xs $ys]
    (match [xs ys] [(list eq) (list eq)]
      {[(loop $i [1 $n]
          {[<cons $x_i #> <cons ,x_i #>]
           @...}
          ![<cons $y _> <cons ,y _>]
        (map (lambda [$i] x_i) (between 1 n))]})))
```

4.4 Loop Patterns in Practice

Loop patterns are used for describing repetitions in a pattern. It is useful when we construct a complicated pattern by combining simple pattern constructors (Sect. 4.4.1) and when the number of pattern variables that appear in a pattern changes by parameters (Sect. 4.4.2). In such situations, very complicated recursion is necessary for describing algorithms. Loop patterns make the descriptions of these algorithms intuitive by confining recursion in a pattern. This section introduces such examples.

4.4.1 Pattern Matching for Trees

This section demonstrates loop patterns by showing pattern matching for trees. The nodes of the trees in this section have an arbitrary number of children as XML, and they are handled as a multiset. A matcher for such a tree is defined as `tree` in Sect. 3.6. We use this matcher in this section.

We describe patterns for a category tree of programming languages. `treeData` defines the category tree. For example, `"Egison"` belongs to the `"Pattern-matching-oriented"` category, and the `"Dynamically typed"` sub-category of the `"Functional programming"` category.

```
(define $tree-data
  <Node "Programming language"
    {<Node "Pattern-matching-oriented"
       {<Leaf "Egison">}>
     <Node "Functional language"
       {<Node "Strictly typed"
          {<Leaf "OCaml"> <Leaf "Haskell"> <Leaf "Curry"> <Leaf "Coq">}>
        <Node "Dynamically typed"
          {<Leaf "Egison"> <Leaf "Lisp"> <Leaf "Scheme"> <Leaf "Racket"> <Leaf "Clojure">}>}>
     <Node "Logic programming"
       {<Leaf "Prolog"> <Leaf "LiLFeS"> <Leaf "Curry">}>
     <Node "Object oriented"
       {<Leaf "C++"> <Leaf "Java"> <Leaf "Ruby"> <Leaf "Python"> <Leaf "OCaml">}>}>)
```

The `matchAll` expression below enumerates all categories to which a specified language belongs. A loop pattern is used to describe a pattern for this purpose because leaves can appear at an arbitrary depth. This pattern is interesting because the ellipsis pattern is not placed the tail of the repeat pattern. The ability to choose the position of expansion of a repeat pattern allows us to apply the loop patterns to trees.

```
(define $ancestors
  (lambda [$x $t]
    (match-all tree-data (tree string)
      [(loop $i [1 $n]
         <node $c_i <cons ... _>>
         <leaf ,x">)
       (map (lambda [$i] c_i) (between 1 n))])))

(ancestors "Egison" treeData)
; {{"Programming language" "Pattern-matching-oriented"} {"Programming language" "Functional language"
     "Dynamically typed"}}
```

It is also possible to enumerate all languages that belong to a specific sub-category. We can use a doubly-nested loop pattern for this purpose because it allows the sub-category to appear at an arbitrary depth. The following pattern matches all the languages that belong to a specified category. We used `matchAllDFS` for this enumeration to make the order of the languages in the result the same as the order with which they appear in the tree.

```
(define $descendants
  (lambda [$x $t]
    (match-all tree-data (tree string)
      [(loop _ [1 _]
         <node _ <cons ... _>>
         <node ,x (loop _ [1 _]
                    <node _ <cons ... _>>
                    <leaf $y>)>)
       y])))

(descendants "Functional language" treeData)
-- {"OCaml" "Haskell" "Curry" "Coq" "Egison" "Lisp" "Scheme" "Racket"}
```

Egison is more elegant than XML path language [3] for handling trees because we can describe the wide range of patterns by just combining a few simple pattern constructors and the loop patterns. In XML path, we would instead have to use the built-in `ancestor` command to enumerates all ancestors of a node, for example.

4.4.2 N-Queen Problem

This section introduces a more tricky example of nested loop patterns by introducing an n-queen solver in Egison. The n-queen problem is the problem of placing n chess queens on an $n \times n$ board such that no queen can attack any of the other queens. In chess, a queen can attack other chess pieces on the same row, column, and diagonal.

Let us start by showing a program for solving the four queen problem. In this program, we represent the positions of the four queens with a list. The number of the n-th element represents the row number of the queen of the n-th line. The solution must be a rearrangement of the list `[1,2,3,4]` because no two queens can be in the same line or row. Therefore, we pattern-match a collection `[1,2,3,4]` as a multiset of integers. The requirement that all two queens must not share the same diagonal is represented with conditions $a_1 \pm 1 \neq a_2$, $a_1 \pm 2 \neq a_3$, $a_2 \pm 1 \neq a_3$, $a_1 \pm 3 \neq a_4$, $a_2 \pm 2 \neq a_4$, and $a_3 \pm 1 \neq a_4$.

```
(define $four-queens
  (match-all {1 2 3 4} (multiset integer)
    [<cons $a_1
      <cons (& !,(- a_1 1) !,(+ a_1 1) $a_2)
       <cons (& !,(- a_1 2) !,(+ a_1 2) !,(- a_2 1) !,(+ a_2 1) $a_3)
        <cons (& !,(- a_1 3) !,(+ a_1 3) !,(- a_2 2) !,(+ a_2 2) !,(- a_3 1) !,(+ a_3 1) $a_4)
         <nil>>>>>
     {a_1 a_2 a_3 a_4}]))

four-queens ; {{2 4 1 3} {3 1 4 2}}
```

We can use a doubly-nested loop pattern for generalizing this pattern for the n-queen solver. The index pattern variable "i" of the outer loop is referred in the index range of the inner loop pattern for describing the difference of the repeat count of inner loop patterns. Also note that the values bound in the previously repeated pattern are referred as "a_j" in `#(a_j - (i - j))` and `!#(a_j + (i - j))`. Non-linearity of indexed pattern variables are effectively used for representing this pattern.

```
(define $n-queens
  (lambda [$n]
    (match-all (between 1 n) (multiset integer)
      [<cons $a_1
        (loop $i [2 n]
              <cons (loop $j [1 (- i 1)]
                          (& !,(- a_j (- i j)) !,(+ a_j (- i j)) ...)
                          $a_i)
                    ...>
              <nil>)>
       (map (lambda [$i] a_i) (between 1 n))])))

(n-queens 4) ; {{2 4 1 3} {3 1 4 2}}
```

5 Pattern-Matching-Oriented Programming in More Practical Situations

This section discusses how pattern-matching-oriented programming changes the implementation of more practical algorithms and software.

5.1 SAT Solver

To see the effect of pattern-matching-oriented programming for implementing practical algorithms, we implement a SAT solver. A SAT solver determines whether a given propositional logic formula has an assignment with which the formula is evaluated to true. Input formulae for SAT solvers are often in *conjunctive normal form*. A formula in conjunctive normal form is a conjunction of clauses, which are disjunctions of *literals*. A literal is a formula whose form is P or $\neg P$. For example, $(P \wedge Q) \vee (\neg P \wedge R) \vee (\neg P \wedge \neg R)$ is a formula in conjunctive normal form that has a solution; $P =$ `False`, $Q =$ `True`, and $R =$ `True`.

5.1.1 The Davis-Putnam Algorithm

In our implementation, propositional logic formulae in conjunctive normal form are represented as a collection of collections of literals. We can pattern-match them as a multiset of multisets of literals because both $\wedge$ and $\vee$ are commutative operators. Therefore, the matcher for these formulae can be defined by simply composing matchers as "`multiset (multiset integer)`".

The program below shows the main part of our implementation of the Davis-Putnam algorithm[14]. The `sat` function takes a list of propositional variables and a logical formula, and returns `True` if there is a solution, otherwise returns `False`.

```
(define $sat
  (lambda [$vars $cnf]
    (match [vars cnf] [(multiset integer) (multiset (multiset integer))]
      {[[_ <nil>] #t]
       [[_ <cons <nil> _>] #f]
       [[_ <cons <cons $l <nil>> _>] ; 1-literal rule
        (sat (delete (abs l) vars) (assign-true l cnf))]
       [[<cons $v $vs> !<cons <cons ,(neg v) _> _>] ; pure literal rule (positive)
        (sat vs (assign-true v cnf))]
       [[<cons $v $vs> !<cons <cons ,v _> _>] ; pure literal rule (negative)
        (sat vs (assign-true (neg v) cnf))]
       [[<cons $v $vs> _] ; otherwise
        (sat vs {@(resolve-on v cnf) @(delete-clauses-with {v (neg v)} cnf)})]})))
```

The first match clause describes the input formula has a solution when it is empty. The second match clause describes that there is no solution when clauses include an empty clause. The third match clause represents *1-literal rule*. When the input formula includes a clause with a single literal, we can assign that literal `True` at once. The fourth match clause describes that if there is a propositional variable that appears only positively, we can set the value of this literal `True` and remove the propositional variable of `x` from the variable list and the clauses that include this literal from the formula. For example, $(p \wedge q) \vee (\neg p \wedge r) \vee (\neg p \wedge \neg r)$ contains a propositional variable q only positively, so we can assign q `True` and remove the first clause from the next recursion. The fifth match clause describes the opposite of the fourth match clause. It removes the clauses that include this literal if there is a propositional variable that appears only negatively (p in the above sample is such propositional variable). The final match clause applies the resolution principle. This match clause lists all pairs of clauses $p \wedge C$ and $\neg p \wedge D$ (let C and D be a disjunction of literals), and obtains new clauses $C \wedge D$.

The above definition of `sat` describes all rules for simplifying an input formula by fully utilizing pattern matching for multisets. In traditional functional languages, we need to call several library functions and define several helper functions to describe these conditional branches. The same algorithm is implemented in OCaml in [14].

5.1.2 Pattern Matching for Resolution

The `resolveOn` function is defined using `matchAll` as follows.

```
(define $resolve-on
  (lambda [$v $cnf]
    (match-all cnf (multiset (multiset integer))
      [<cons <cons ,v $xs>
        <cons <cons ,(neg v) $ys>
          _>>
       (unique (filter (lambda [$c] (not (tautology? c))) {@xs @ys}))])))
```

The above `resolveOn` removes tautological clauses by using the `tautology` predicate. We can remove the use of this function by modifying the above pattern using a sequential not-patterns discussed in Sect. 4.3.

```
(define $resolve-on
  (lambda [$v $cnf]
    (match-all cnf (multiset (multiset integer))
      [{<cons <cons ,v (& # $xs)>
         <cons <cons ,(neg v) (and # $ys)>
          _>>
```

```
        ![<cons $l _> <cons ,(neg l) _>]}
       (unique {@xs @ys})])))
```

5.2 Computer Algebra

As an application of Egison, we have developed a computer algebra system in Egison [10]. In Egison, we can implement a pattern-matching engine for mathematical expressions in a short program by defining a matcher for mathematical expressions.

```
mathExpr = matcher
             Div $ $ as (mathExpr, mathExpr) with
               Div $x $y -> [(x, y)]
               $tgt -> [(tgt, 1)]
             Poly $ as multiset mathExpr with
               $tgt -> [tgt]
             Term $ $ as (integer, multiset (mathExpr, integer)) with
               Term $c $xs -> [(c, xs)]
             App $ $ as (mathExpr, list mathExpr) with
               App $f $args -> [(f, args)]
             Symbol $ as string with
               Symbol $name -> [name]
             $ as something with
               $tgt -> [tgt]
```

The matcher for mathematical expressions is used for implementing simplification algorithms of mathematical expressions. For example, a program that simplifies a mathematical expression by reducing $\cos^2(\theta)+\sin^2(\theta)$ to 1 can be implemented as follows.

```
simplifyCosAndSinInPoly poly =
  match poly as mathExpr with
    Poly (Term $n ((App (Sym #"cos") [$x]), 2):$y) : Term #n ((App (Sym #"sin") [#x], 2):#y) : r) ->
     simplifyCosAndSinInPoly (n * y + r)
    _ -> poly
```

Using some syntax sugar, we can rewrite the above pattern as follows.

```
simplifyCosAndSinInPoly poly =
  match poly as mathExpr with
    (+ (* $n (#cos $x)^#2 $y) (* #n (#sin $x)^#2 #y) $r) ->
     simplifyCosAndSinInPoly (n * y + r)
    _ -> poly
```

The definition of a matcher for mathematical expressions is simple compared with the pattern-matching engines of the other computer algebra systems. As a result, the implementation of the whole computer algebra system is also compact and straightforward; therefore, this computer algebra system is easily extensible. This extensibility allows us to experiment with new features easily.

This extensibility is a significant advantage in the future of computer algebra system in the field of which there are still notations that are popular among researchers but cannot be used in programs. There are many possibilities of research for extending computer algebra systems to support these notations. The extensibility of computer algebra systems will help us advance this research.

Such work has already been done by Egi who has developed a natural method for importing tensor index notation into programming [8]. Thanks to this work, Egison became an appropriate computer algebra system for describing formulae of differential geometry. We are now preparing a paper whose results are calculated using Egison collaborating with researchers of differential geometry.

5.3 Database Query Languages

Egison pattern matching can provide a unified query language for various kinds of databases. For example, let us consider an SNS database and a query to list all users who are followed by a user whose name is

"Egison_Lang" but who do not follow this user. This query can be written with matchAll as follows using Egison pattern matching. This query pattern-matches the tables of a relational database as sets.

```
matchAll (users, follows, users) as (set user, set follow, set user) with
  ((and (Name #"Egison_Lang") (ID $uid)) : _,
   (and (FromID #uid) (ToID $fid)) : !((and (FromID #fid) (ToID #uid)) : _),
   (and (ID #fid) (Name $fname)) : _) -> (fid, fname)
```

The conciseness of the queries is an important advantage of Egison over other query languages. For example, the same query described in SQL is more complicated. We need to write all conditions in WHERE clauses instead of a non-linear pattern and a sub-query instead of a not-pattern. A query in the pattern-matching-oriented style can be interpreted by reading it once from left to right in order, whereas one in SQL cannot.

```
SELECT DISTINCT ON (user.name) user.name
  FROM user AS user1, follow AS follow1, user AS user2
  WHERE user1.name = 'Egison_Lang' AND follow1.from_id = user1.id AND user2.id = follow1.to_id
    AND NOT EXISTS
      (SELECT '' FROM follow AS follow2
         WHERE follow2.from_id = follow1.to_id AND follow2.to_id = user1.id
```

In addition to the conciseness, its generality is another advantage of Egison. Query languages such as SQL [6], XML path language [3] and graph query languages [19, 21, 24] only focus on handling their target data structures and have many built-in functions to handle various patterns. On the other hand, Egison pattern-matching system allows users to describe various patterns for various data types in a unified manner with a small number of pattern constructors.

6 Conclusion

This paper has presented programming techniques that replace explicit recursions for traversing search trees with intuitive patterns and allow programmers to concentrate on the description of the essential parts of algorithms that reduce the computational complexities of the algorithms. This paper listed many algorithms that can be represented more elegantly this way. These include many very basic list processing functions to some larger more practical algorithms.

We believe the development of these programming techniques that enable the more intuitive representation of algorithms extends the limit on the complexity of software that we can practically implement and has potential to accelerate research of computer science as a whole. We focused on the data abstraction for non-free data types in this paper. However, there must be more data types waiting to be abstracted. We hope this paper leads to the further evolution of pattern matching and progress of data abstraction in the next age.

Acknowledgments

We thank Kentaro Honda, who is the first real user of Egison, for his contribution to some sample programs in this paper. We thank Pierre Imai, Matthew Roberts, and Paul Harvey for helpful feedback on the earlier versions of the paper.

References

[1] Sergio Antoy. "Constructor-based conditional narrowing". In: *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2001. DOI: 10.1145/773184.773205.

[2] Sergio Antoy. "Programming with narrowing: A tutorial". In: *Journal of Symbolic Computation* 45.5 (2010). DOI: 10.1016/j.jsc.2010.01.006.

[3] Anders Berglund et al. "XML Path Language (XPath)". In: *World Wide Web Consortium (W3C)* (2003).

[4] Rod M. Burstall. "Proving Properties of Programs by Structural Induction". In: *The Computer Journal* 12.1 (1969), pp. 41–48. DOI: 10.1093/comjnl/12.1.41.

[5] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. “HOPE: An Experimental Applicative Language”. In: *Proceedings of the 1980 ACM conference on LISP and functional programming*. ACM. 1980, pp. 136–143. DOI: 10.1145/800087.802799.

[6] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A structured English query language”. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 1974. DOI: 10.1145/800296.811515.

[7] Satoshi Egi. “Loop Patterns: Extension of Kleene Star Operator for More Expressive Pattern Matching against Arbitrary Data Structures”. In: *The Scheme and Functional Programming Workshop*. 2018.

[8] Satoshi Egi. “Scalar and Tensor Parameters for Importing Tensor Index Notation including Einstein Summation Notation”. In: *The Scheme and Functional Programming Workshop*. 2017.

[9] Satoshi Egi and Yuichi Nishiwaki. “Non-linear Pattern Matching with Backtracking for Non-free Data Types”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2018, pp. 3–23. DOI: 10.1007/978-3-030-02768-1_1.

[10] *Egison Mathematics Notebook*. https://www.egison.org/math. [Online; accessed 2019-02-21]. 2016.

[11] Martin Erwig. “Active patterns”. In: *Implementation of Functional Languages* (1996). DOI: 10.1007/3-540-63237-9_17.

[12] Michael Hanus. “Multi-paradigm declarative languages”. In: *International Conference on Logic Programming*. 2007.

[13] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. “Curry: A truly functional logic language”. In: *Proc. ILPS*. Vol. 95. 5. 1995, pp. 95–107.

[14] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511576430.001.

[15] Paul Hudak et al. “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, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856.

[16] F. V. McBride, D. J. T. Morrison, and R. M. Pengelly. “A symbol manipulation system”. In: *Machine Intelligence* 5 (1969).

[17] John McCarthy. “Symbol Manipulating Language - Revisions of the Language”. In: *MIT AI Lab. AI Memo No. 4* (1958).

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1991, pp. 124–144. DOI: 10.1007/3540543961_7.

[19] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and Complexity of SPARQL”. In: *International semantic web conference*. 2006.

[20] Christian Queinnec. “Compilation of Non-linear, Second Order Patterns on S-Expressions”. In: *Programming Language Implementation and Logic Programming*. 1990. DOI: 10.1007/BFb0024194.

[21] Marko A. Rodriguez. “The Gremlin Graph Traversal Machine and Language”. In: *Proceedings of the 15th Symposium on Database Programming Languages*. 2015. DOI: 10.1145/2815072.2815073.

[22] Olin Shivers. *SRFI 1: List library*. 1999.

[23] *The Egison Programming Language*. https://www.egison.org. [Online; accessed 2019-05-21]. 2011.

[24] *The Neo4j Manual v2.3.3*. https://neo4j.com/docs/stable/index.html. [Online; accessed 14-June-2018]. 2016.

[25] Simon Thompson. “Lawful functions and program verification in Miranda”. In: *Science of Computer Programming* 13.2-3 (1990). DOI: 10.1016/0167-6423(90)90070-T.

[26] Simon Thompson. “Laws in Miranda”. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. 1986. DOI: 10.1145/319838.319839.

[27] Mark Tullsen. “First Class Patterns”. In: Springer, 2000. DOI: 10.1007/3-540-46584-7_1.

[28] David Turner. “Miranda: A non-strict functional language with polymorphic types”. In: *Functional programming languages and computer architecture*. 1985. DOI: 10.1007/3-540-15975-4_26.

[29] David Turner. “Some History of Functional Programming Languages”. In: *International Symposium on Trends in Functional Programming*. Springer. 2012, pp. 1–20. DOI: 10.1007/978-3-642-40447-4_1.

[30] Philip Wadler. “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*. 1987. DOI: 10.1145/41625.41653.

A Creating Your Own Matchers

There are data types whose matchers cannot be defined by just composing the existing matchers. For example, matchers for lists and multisets are such matchers. This section explains how to define matchers from scratch through examples. We omit an explanation of the formal semantics of matchers and the pattern-matching algorithm inside Egison and instead focus on an explanation of programming techniques used for defining matchers. For the detailed description of the formal semantics, please refer to the original paper of Egison pattern matching [9].

Sect. A.1 explain how to create a matcher by showing a definition of the `multiset` matcher. We show that the pattern-matching-oriented programming techniques presented in this paper are useful also for matcher definitions. Sect. A.2 introduces a programming technique specific to matcher definitions by showing a definition of the `sorted-list` matcher.

A.1 Multiset Matcher

Matcher definition is the most technical part in pattern-matching-oriented programming. This section explains the method for defining a matcher by showing a matcher definition for multisets. The program below shows a matcher definition for multisets. The multiset matcher is the most basic nontrivial matcher. This section explores the detail of this definition.

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

A matcher is defined using the `matcher` expression. The `matcher` expression is a built-in syntax of Egison. `matcher` takes a collection of *matcher clauses*. A matcher clause is a triple of a *primitive-pattern pattern*, a *next-matcher expression*, and a *next-target expression*.

A matcher is a kind of function that takes a pattern and target, and returns lists of the next *matching atoms*. A matching atom is a triple of a pattern, target, and matcher. A primitive-pattern pattern matches a pattern. Patterns that match with *pattern holes* ("$" inside primitive-pattern patterns) are next patterns. A next-matcher expression returns next matchers. A next-target expression is a function that takes a target and returns a list of next targets. A matcher generates a list of the next matching atoms by combining next patterns, next matchers, and a list of next targets.[7]

The `multiset` matcher has four matcher clauses. The first matcher clause handles a nil pattern, and it checks whether the target is an empty collection or not. The second matcher clause handles a cons pattern. The third matcher clause handles a value pattern. This matcher clause defines the equality of multisets. The fourth matcher clause handles the other patterns for multiset: a pattern variable and wildcard.

First, we focus on the second matcher clause. The primitive-pattern pattern of the second matcher clause is "`$ : $`", and the next matcher expression is "`(a, multiset a)`". It means two arguments of the cons

[7]In Egison, pattern matching is implemented as reductions of stacks of matching atoms. Each list of the next matching atoms returned by a matcher 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 single 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.

pattern are next patterns and they are pattern-matched using the “a” and “multiset a” matchers, respectively. “a” is an argument of multiset and the matcher for inner elements of a multiset. In the next target expression, a simple join-cons pattern is used to decompose a target collection into an element and the rest collection. For example, when the target is a collection [1,2,3], this next-target expression returns [(1,[2,3]),(2,[1,3]),(3,[1,2])]. Each tuple of the next targets is pattern-matched using the next patterns and the next matchers recursively. For example, 1 and [2,3] are pattern-matched using the “a” and “multiset a” matcher with the first and the second argument of the cons pattern, respectively.

Next, we focus on the third matcher clause. This matcher clause is as technical as the second matcher clause. The primitive-pattern pattern of this matcher clause is “#$val”. It is called a *value-pattern pattern*. A value-pattern pattern matches a value pattern. This matcher clause compares the content of a value pattern (val) and a target (tgt) as multisets. The match expression is used for this comparison. Interestingly, tgt is recursively pattern-matched as “multiset a”.

The first and the third match clauses of this match expression are simple. The first match clause describes that it returns “[()]” when both val and tgt are empty. This return value means pattern matching for the value pattern succeeded. The third match clause describes that it returns “[]” if pattern matching for the patterns of both the first and the second match clause failed. This return value means pattern matching for the value pattern failed.

The second match clause is the most technical part of this match expression. The value pattern “#xs” is recursively pattern-matched using this matcher clause itself. The collection xs is one element shorter than tgt. Therefore, this recursion finally reaches the first or the third match clause if val and tgt are finite.

Finally, let us also explain the fourth matcher clause. This matcher clause creates the next matching atom by just changing the matcher from “multiset a” to something.

A.2 Matcher for Sorted Lists

Modularization of pattern-matching algorithms by matchers not by patterns enables polymorphic patterns. However, its merit extends beyond polymorphic patterns; matchers enable descriptions of more efficient pattern matching keeping patterns concise. The reason is that pattern matching against patterns inside matcher definitions allows us to describe more detailed pattern-matching algorithms. This section shows such an example, a matcher for sorted lists.

The program that used a doubly-nested join-cons pattern for enumerating pairs of prime numbers whose forms are $(p, p + 6)$ gets slower when the number of the enumerating prime pairs gets larger. The reason is that the program enumerates all the combinations of prime numbers. For example, the program tries to match all the pairs such as $(3, 5), (3, 7), (3, 11), (3, 13), (3, 17), (3, 19)$, and so on. However, we should avoid enumerating the pairs after $(3, 11)$, the first pair whose difference is more than 6. This is because it is obvious that the difference between all the pairs after $(3, 11)$ are more than 6.

```
(take 10 (match-all primes (sorted-list integer)
           [<join _ <cons $p _ <cons ,(+ p 6) _>>>> [p (+ p 6)]]))
; {[5 11] [7 13] [11 17] [13 19] [17 23] [23 29] [31 37] [37 43] [41 47] [47 53]}
```

We can avoid this unnecessary search by creating a new matcher that is specialized for sorted lists. We can define such a matcher by adding a matcher clause with the primitive-pattern pattern “$ ++ #$px : $” to the list matcher as shown below. This matcher clause improves the theoretical time complexity of the above pattern from $O(n^2)$ to $O(n)$.

```
(define $sorted-list
  (lambda [$a]
    (matcher
      {[<join $ <cons ,$px $>> [(sorted-list a) (sorted-list a)]
        {[$tgt (match-all tgt (list a)
                 [(loop $i [1 $n] <cons (& ?(lt? $ px) $h_i) ...> <cons ,px $ts>)
                  [(map (lambda [$i] h_i) (between 1 n)) ts]])]}]
       ...})))
```

Note that this method is only applicable to Egison that modularizes pattern-matching methods for each matcher, not for each pattern. The reason is that we need to match patterns whose form is “_ ++ #x : _”

as mentioned above. If pattern-matching methods are modularized for each pattern, we need to introduce a new pattern constructor "`joinCons ... ...`" that is equivalent to "`_ ++ ... : ...`" for this purpose.

B Implementation of a SAT Solver

```
(define $delete-literals
  (lambda [$ls $cnf]
    (map (lambda [$c] (match-all [c ls] [(multiset integer) (multiset integer)]
                        [[<cons $l _> !<cons ,l _>] l]))
         cnf)))

(define $delete-clauses-with
  (lambda [$ls $cnf]
    (match-all [ls cnf] [(multiset integer) (multiset (multiset integer))]
      [{[# <cons (& # $c) _>]
        ![<cons $l _> <cons ,l _>]}
       c])))

(define $assign-true
  (lambda [$l $cnf]
    (delete-literals {(neg l)} (delete-clauses-with {l} cnf))))

(define $resolve-on
  (lambda [$v $cnf]
    (match-all cnf (multiset (multiset integer))
      [{<cons <cons ,v (& # $xs)>
         <cons <cons ,(neg v) (and # $ys)>
          _>>
        ![<cons $l _> <cons ,(neg l) _>]}
       (unique {@xs @ys})])))

(define $sat
  (lambda [$vars $cnf]
    (match [vars cnf] [(multiset integer) (multiset (multiset integer))]
      {[[_ <nil>] #t]
       [[_ <cons <nil> _>] #f]
       [[_ <cons <cons $l <nil>> _>] (sat (delete (abs l) vars) (assign-true l cnf))]
       [[<cons $v $vs> !<cons <cons ,(neg v) _> _>] (sat vs (assign-true v cnf))]
       [[<cons $v $vs> !<cons <cons ,v _> _>] (sat vs (assign-true (neg v) cnf))]
       [[<cons $v $vs> _] (sat vs {@(resolve-on v cnf) @(delete-clauses-with {v (neg v)} cnf)})]})))

(sat {1} {{1}}) ; #t
(sat {1} {{1} {-1}}) ; #f
(sat {1 2 3} {{1 2} {-1 3} {1 -3}}) ; #t
(sat {1 2} {{1 2} {-1 -2} {1 -2}}) ; #t
(sat {1 2} {{1 2} {-1 -2} {1 -2} {-1 2}}) ; #f
(sat {1 2 3 4 5} {{-1 -2 3} {-1 -2 -3} {1 2 3 4} {-4 -2 3} {5 1 2 -3} {-3 1 -5} {1 -2 3 4} {1 -2 -3
    5}}) ; #t
(sat {1 2} {{-1 -2} {1}}) ; #t
```