New Generation Computing (2022) 40:481–506
https://doi.org/10.1007/s00354-022-00177-z

NEW GENERATION COMPUTING

Embedding Non-linear Pattern Matching with Backtracking for Non-free Data Types into Haskell

Satoshi Egi[1] · **Akira Kawata**[2,3] · **Mayuko Kori**[4,5] · **Hiromi Ogawa**[6,7]

Received: 8 August 2021 / Accepted: 25 May 2022 / Published online: 21 July 2022

Abstract

Pattern matching is an important language construct for data abstraction. Many pattern-match extensions have been developed for extending the range of data types to which pattern matching is applicable. Among them, the pattern-match system proposed by Egi and Nishiwaki features practical pattern matching for non-free data types by providing a user-customizable non-linear pattern-match facility with backtracking. However, they implemented their proposal only in dynamically typed programming languages, and there were no proposals that allow programmers to benefit from both static type systems and expressive pattern matching. This paper proposes a method for implementing this pattern-match facility by meta-programming in Haskell. There are two technical challenges: (i) we need to design a set of typing rules for the pattern-match facility; (ii) we need to embed these typing rules in Haskell to make types of the pattern-match expressions inferable by the Haskell type system. We propose a set of typing rules and show that several GHC extensions, such as multi-parameter type classes, datatype promotion, GADTs, existential types, and view patterns, play essential roles for embedding these typing rules into Haskell. The implementation has already been distributed as a Haskell library *miniEgison* via Hackage.

Keywords Pattern matching · Non-free data types · Type system · Haskell

✉ Satoshi Egi
satoshi.egi@rakuten.com

[1] Rakuten, Inc., Tokyo, Japan

[2] Kyoto University, Kyoto, Japan

[3] Present Address: Preferred Networks, Inc., Tokyo, Japan

[4] The University of Tokyo, Tokyo, Japan

[5] Present Address: National Institute of Informatics, Tokyo, Japan

[6] Koishikawa Secondary Education School, Tokyo, Japan

[7] Present Address: University of Tsukuba, Tsukuba, Japan

Mathematics Subject Classification MSC code1 · MSC code2 · More

1 Introduction

Pattern matching is an important feature of programming languages for intuitive descriptions of algorithms. Pattern matching allows us to replace many verbose function applications for decomposing data (such as **`head`** and **`tail`**) to an intuitive pattern. The pattern-match facility for algebraic data types is common in modern functional programming languages. However, some data types are not algebraic data types, and we cannot use the common pattern-match 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-match extensions have been proposed for extending the range of data types to which pattern matching is applicable to non-free data types [7, 18]. Among them, the pattern-match system proposed by Egi and Nishiwaki [3] features practical pattern matching for non-free data types with the following three features: (i) non-linear pattern matching with backtracking; (ii) user-customizable pattern-match algorithms; (iii) ad hoc polymorphism of patterns. This pattern-match system is first implemented in an interpreter of their proof-of-concept programming language Egison.

The Egison pattern-match facility is also implemented in the existing programming languages. Egi proposed a method for embedding the Egison pattern-match facility in Scheme and implemented his proposal as Scheme macros [2]. These Scheme macros translate Egison pattern-match expressions to functional programs without Egison pattern-match expressions. This Scheme implementation is more than ten times faster than the original Egison interpreter because the Scheme interpreter is more optimized. The next challenge is embedding the Egison pattern-match facility in statically typed languages. There are two technical challenges for this purpose. First, we need to define typing rules for Egison pattern-match expressions. Second, we need to embed these typing rules in Haskell to make types of the pattern-match expressions inferable by Haskell. We cannot directly apply the method taken by the Scheme macros for Haskell, because we cannot simply type these macros. Especially, non-linear patterns and user-customizable patterns make static typing difficult.

This paper introduces a Haskell library that provides the pattern-match facility of Egison. The library has already been distributed via Hackage as *miniEgison*. We developed a method for translating Egison pattern-match expressions to Haskell programs that are automatically type-inferable and compilable by GHC. GHC extensions such as multi-parameter type classes, GADTs, and existential types play essential roles for this purpose. This paper introduces Haskell programming techniques for implementing the proposed library.

1.1 MiniEgison

This section briefly introduces the usage of miniEgison. By introducing miniEgison, we also present the pattern-match facility of Egison.

The **matchAll** expression is the most basic pattern-match expression provided by miniEgison. The **matchAll** expression collects all the pattern-match results and returns a list of the results evaluating the body expression for each pattern-match result. The **matchAll** expression takes three arguments: a target, a matcher, and match clauses. A matcher is an Egison specific object that specifies pattern-match methods. In this case, the matcher **(List Something)** specifies that the pattern is interpreted as a pattern for a list of something. The **Something** matcher is the only built-in matcher that can handle only a wildcard or a pattern variable but can pattern-match arbitrary objects. A match clause is constructed using the quasi-quote [16] provided by Template Haskell [15]. The **mc** quasi-quoter is defined in the proposed library to desugar a match clause. The **matchAll** expression below pattern-matches the list **[1, 2, 3]** with the pattern **$x : $xs** as a list, and returns the collection of the tuple **(x, xs)**. Patterns that start with **$** (**$x** and **$xs** in this sample) are pattern variables. The evaluation result of the **matchAll** expression contains only one element because the cons pattern for a list has only one decomposition.

```
matchAll [1, 2, 3] (List Something)
  [[mc| $x : $xs -> (x, xs) |]]
-- [(1, [2, 3])]
```

The **matchAll** expression takes a list of match clauses and can handle multiple match clauses. When there are multiple match clauses, **matchAll** concatenates the results for each match clause. "**matchAll** *t m* [c_1,c_2,...]" is equivalent to "**matchAll** *t m* [c_1] **++ matchAll** *t m* [c_2] **++** ...".

```
matchAll [1, 2, 3] (List Something)
  [[mc| $x : $xs -> (x, xs) |]
  ,[mc| _ : $x : $xs -> (x, xs) |]]
-- [(1, [2, 3]), (2, [3])]
```

Matchers, the second argument of **matchAll**, play an important role in achieving ad hoc polymorphism of patterns. In the following code, the matcher is changed from **(List Something)** to **(Multiset Something)**. The cons pattern for a multiset is defined to decompose a target list into an arbitrary element and the rest elements. Therefore, the pattern-match results changes as follows:

```
matchAll [1, 2, 3] (Multiset Something)
  [[mc| $x : $xs -> (x, xs) |]]
-- [(1, [2, 3]), (2, [1, 3]), (3, [1,2])]
```

Ad hoc polymorphism of patterns decreases the number of pattern constructor's names we need to remember. Users can define pattern-match algorithms of each pattern for each matcher. We call this feature customizability of pattern-match algorithms. Achieving both customizability and ad hoc polymorphism of patterns together is the first main technical challenge of the paper.

Matching in miniEgison allows non-linear patterns, where the same variable can have multiple occurrences in a single pattern. The pattern that starts with **#** is called a *value pattern*. An arbitrary Haskell expression follows after **#**. A value pattern checks whether the target and the content of the value pattern are equal or not. The **Eql** matcher is defined to use **==** for checking the equality. The pattern below matches when there are pairs of sequential elements. There are two sequential pairs: **(1, 2)** and **(4, 5)**. Value patterns can refer to the values bound to the pattern variables that appear on the left-side of patterns. For example, the pattern **#(x + 1) : $x : _** is invalid. This restriction makes patterns readable from left to right in order and makes the internal pattern-match algorithm simple.

```
matchAll [1, 5, 2, 4] (Multiset Eql)
  [[mc| $x : #(x + 1) : _ -> (x, x + 1) |]]
-- [(1, 2), (4, 5)]
```

The type of the expression inside a value pattern is inferable by the GHC type checker. Maintaining non-linear patterns in a static type system of Haskell is a major technical challenge of the paper.

When a variable occurring in a value pattern has no corresponding pattern variable on its left of the pattern, it is bound by a Haskell variable. For example, the value pattern **#x** refers to the value bound by the first argument of **delete** in the following code.

```
delete x xs =
  match xs (List Eql)
    [[mc| $hs ++ #x : $ts -> hs ++ ts |],
     [mc| _ -> xs |]]

delete 2 [1, 2, 3, 4]
-- [1, 3, 4]
```

The pattern-match algorithm inside miniEgison uses backtracking for traversing a search tree as we will explain in Sect. 4.1. The following expressions search the sequential pairs and triples from the list that contains n 0s. Thanks to the backtracking mechanism, the time complexities for both expressions are quadratic. This is because pattern matching for **#(x + 1)** always fails and pattern matching for **#(x + 2)** in the second expression is never tried. Non-linear patterns have an advantage over pattern

guards that they can skip unnecessary matching candidates thanks to backtracking. On the other hand, pattern guards fail to filter out those unnecessary candidates.

```
matchAll (replicate 0 n) (Multiset Eql)
  [[mc| $x : #(x + 1) : _ -> x |]]
-- returns [] in O(n^2)
matchAll (replicate 0 n) (Multiset Eql)
  [[mc| $x : #(x + 1) : #(x + 2) : _ -> x |]]
-- returns [] in O(n^2)
```

The **matchAll** expression is designed to be able to enumerate all the infinitely many pattern-match results by traversing a search tree in the breadth-first order. The **cons** pattern for the **Set** matcher that appears in the following code decomposes a list to an arbitrary element and the original list itself. If we regard a set as a collection that contains infinitely many copies of each element, this specification of the **cons** pattern for a set is natural. This specification is also useful, for example, when pattern-matching graphs as sets of edges.

```
take 10 (matchAll [1..] (Set Something)
  [[mc| $x : $y : _ -> (x, y) |]])
-- [(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3), (3, 2), (4,
    ↪ 1)]
```

On the other hand, **matchAllDFS** traverses a search tree for pattern matching in the depth-first order. If we change the above **matchAll** to **matchAllDFS**, the order of the pattern-match results changes as follows:

```
take 10 (matchAllDFS [1..] (Set Something)
  [[mc| $x : $y : _ -> (x, y) |]])
-- [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1,
    ↪ 10)]
```

MiniEgison also provides **match** and **matchDFS**, which calculate only the first pattern-match result. The **match** expression is simply implemented as follows (**matchDFS** is defined analogously):

```
match tgt m cs = head (matchAll tgt m cs)
```

To keep the code simple, we made use of **head** to implement **match**. On failure of pattern-match, this raises an uninformative exception, but it can be processed by an error handler.

Pattern matching for poker hand shown in Fig. 1 is a typical application of Egison pattern matching. We can represent each poker hand in a single pattern by pattern matching a list of cards as a multiset. We can compile the program below using GHC. The **CardM** matcher is a matcher for playing cards. We will define **CardM** in Sect. 7.2. "**[**p_1 **,** p_2 **,** ... **,** p_n **]**" is syntactic sugar of a pattern whose form is "p_1 **:** p_2 **:** ... **:** p_n **:** **[]**".

```
data Suit = Spade | Heart | Club | Diamond deriving (Eq)
data Card = Card Suit Integer

poker :: [Card] -> String
poker cs = matchDFS cs (Multiset CardM)
 [[mc| [card $s $n, card #s #(n-1), card #s #(n-2), card #s #(n-3), card #s #(n
     ↪ -4)] -> "Straight flush" |],
  [mc| [card _ $n, card _ #n, card _ #n, card _ #n, _] -> "Four of a kind" |],
  [mc| [card _ $m, card _ #m, card _ #m, card _ $n, card _ #n] -> "Full house"
     ↪ |],
  [mc| [card $s _, card #s _, card #s _, card #s _, card #s _] -> "Flush" |],
  [mc| [card _ $n, card _ #(n-1), card _ #(n-2), card _ #(n-3), card _ #(n-4)]
     ↪ -> "Straight" |],
  [mc| [card _ $n, card _ #n, card _ #n, _, _] -> "Three of a kind" |],
  [mc| [card _ $m, card _ #m, card _ $n, card _ #n, _] -> "Two pair" |],
  [mc| [card _ $n, card _ #n, _, _, _] -> "One pair" |],
  [mc| _ -> "Nothing" |]]
```

Fig. 1 Pattern matching for determining poker hands

In Egison, the users are allowed to define custom pattern-match algorithms for each specific pattern. For example, the cons patterns for multisets can be defined by the users. The only built-in pattern-match algorithms are those for wildcards and pattern variables. The method for defining pattern-match algorithms is explained in Sect. 5.

Thanks to the pattern-match facility, a broader range of algorithms can be given in a declarative way. For example, we can implement **`intersect`** using pattern matching for sets as follows:

```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys =
  matchAll (xs, ys) (Pair (Set Eql) (Set Eql))
    [[mc| ($x : _, #x : _) -> x |]]
```

Though **`intersect`** can be defined in the following shorter code than the above without pattern matching, the above program is more declarative in the sense that the two lists are treated as sets directly.

```
intersect xs ys =  [x | x <- xs, any (== x) ys]
```

The programming techniques utilizing this pattern-match facility are classified as pattern-match-oriented programming design patterns in our previous paper [4].

Currently, miniEgison provides a subset of the functionalities of the original Egison. The original Egison provides non-standard pattern constructs, such as loop patterns, indexed pattern variables, and sequential patterns, that are still not implemented in miniEgison. Loop patterns and indexed pattern variables are used for representing the repetitions inside a pattern. Sequential patterns are used controlling the order of pattern matching. These pattern constructs are introduced in our previous paper [4].

1.2 Related Work

Over the decades, several pattern-match extensions have been proposed and implemented in Haskell. Views [19] are an early such extension that provides users with the user-customizable pattern matching facility. Views allow the users to define custom pattern-match algorithms for each pattern constructor. However, views support neither non-linear patterns nor pattern matching with backtracking. Active patterns [5] provides the user-customizable pattern matching facility with non-linear patterns. However, active patterns do not support backtracking and thus fail to fully utilize the expressiveness of non-linear patterns. First class patterns [17] provides the user-customizable pattern matching with backtracking. However, first-class patterns do not support non-linear patterns. The pattern-match system of Egison [3] can be regarded as an extension that supports all the features proposed above: user-customizable pattern-match algorithms; non-linear patterns; pattern matching with multiple results.

Implementations of some of the above pattern-match extensions are available as open-source and compatible with the latest GHC. Views are implemented as a GHC extension. First-class patterns are implemented as a Haskell library by Pope and Yorgey and distributed via Hackage [14]. In implementing miniEgison, we heavily reuse the ideas from the implementation of this library: a pattern is defined as a function that takes a target and returns a pattern-match result, which is represented by a heterogeneous list [10].

This paper is not the first attempt to implement the Egison pattern-match facility as a library of a mainstream functional programming languages. Egi developed a Scheme library for the Egison pattern-match facility [2]. We reuse many ideas from the implementation of this Scheme library in this paper. For example, the methods for translating **`matchAll`** and non-linear patterns in Scheme and Haskell are very similar. The major contribution of this paper is the mechanism for embedding the translated programs in the statically typed language of Haskell, where we utilize some advanced GHC features for typing non-linear patterns, polymorphic patterns, and matcher.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Sect. 2, we design a set of typing rules for **`matchAll`**, matchers, and patterns. In Sect. 3, we explain how we represent these typing rules in Haskell by showing the type of **`matchAll`**, matchers, and pattern in miniEgison. In Sect. 4, we implement the internal pattern-match procedure of the Egison pattern-match system in Haskell. In Sect. 5, we explain how we can define matchers in miniEgison. In Sect. 6, we compare miniEgison with the Scheme library of Egison and discuss how Egison patterns are embedded in the statically typed language of GHC. In Sect. 7, we discuss the techniques for improving the compatibility with the Haskell patterns. In Sect. 8, we compare the execution performance of miniEgison to that of the original Egison interpreter by showing the benchmarking results. Section 9 concludes the paper.

2 Typing Rules

This section shows a set of typing rules that we designed for Typed Egison [9], a variation of Egison with a static type system. As the original Egison is a dynamically typed programming language, we need to devise typing rules for Egison's non-linear patterns with ad hoc polymorphism.

Figure 2 shows the set of typing rules. In the rules, meta-variables x, e, p, and C denote a variable, an expression, a pattern, and a pattern constructor, respectively. We write T and S to denote types, Γ and Δ to denote type environments and ϵ to denote an empty type environment. We also write $[T]$ to denote a type of a list of T. **`Matcher`** and **`Pattern`** are built-in type constructors: **`Matcher`** is a type of a matcher for T; **`Pattern`** is a type of a pattern for T.

The type judgement $\Gamma \vdash e : T$ states that e has the type T under the type environment Γ. The other type judgment $\Gamma;\Delta \vdash p : T;\Delta'$ states that the pattern p is to be matched against a value of type T under the type environment $\Gamma;\Delta$, where the additional environment Δ gives type assignment on pattern variables that occur on the left of p.[1] This judgement states that p has the type T under the type environment Γ and Δ and also that Δ is updated to Δ'. Δ is an additional type environment that describes the types of pattern variables occuring in the left of p. The comma separated pair Γ, Δ concatenates the two type environments Γ and Δ, where type assignments x in Δ may override those in Γ.

We explain the typing rules for patterns, focusing on the type environments Δ for patterns. The type environment Δ keeps track of types of pattern variables in non-linear patterns. The rule (T-PatternVariable) adds a type binding for each pattern variable. The assignments in Δ are used to type-check value patterns

A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \texttt{Matcher}\ T_1 \qquad \Gamma;\epsilon \vdash p : \texttt{Pattern}\ T_1;\Delta \qquad \Gamma,\Delta \vdash e_3 : T_3}{\Gamma \vdash \texttt{matchAll}\ e_1\ \texttt{as}\ e_2\ \texttt{with}\ p\ \texttt{->}\ e_3 : [T_3]}\ \text{T-MatchAll}$$

Typing rules for patterns

$$\frac{}{\Gamma;\Delta \vdash \_ : \texttt{Pattern}\ T;\Delta}\ \text{T-Wildcard} \qquad \frac{\Gamma,\Delta \vdash e : T}{\Gamma;\Delta \vdash \#e : \texttt{Pattern}\ T;\Delta}\ \text{T-ValuePattern}$$

$$\frac{}{\Gamma;\Delta \vdash \$x : \texttt{Pattern}\ T;\Delta,(x:T)}\ \text{T-PatternVariable}$$

$$\frac{\begin{array}{c}\Gamma \vdash C : (\texttt{Pattern}\ S_1,\cdots,\texttt{Pattern}\ S_n) \to \texttt{Pattern}\ T \\ \Gamma;\Delta_0 \vdash p_1 : \texttt{Pattern}\ S_1;\Delta_1 \qquad \Gamma;\Delta_1 \vdash p_2 : \texttt{Pattern}\ S_2;\Delta_2 \\ \cdots \qquad \Gamma;\Delta_{n-1} \vdash p_n : \texttt{Pattern}\ S_n;\Delta_n\end{array}}{\Gamma;\Delta_0 \vdash (C\ p_1\ p_2\ \ldots\ p_n) : \texttt{Pattern}\ T;\Delta_n}\ \text{T-ConstructorPattern}$$

Fig. 2 Typing rules for **`matchAll`** and patterns of Typed Egison

[1] We borrow this notation from [8].

(T-ValuePattern). The type environment Δ is updated and passed in a pattern in order from left to right (T-ConstructorPattern).

In Fig. 2, we do not define the behaviour for the case where a single pattern contains multiple occurrences of the same pattern variable. This is because we cannot find a useful example that contains such a pattern. Currently, we implement miniEgison to override the binding on variable x, if $x \in dom(\Delta)$ in T-PatternVariable. A value pattern **#x** refers to the value matched by the closest preceding pattern variable **$x**.

In implementing miniEgison, we need to embed these typing rules in Haskell, which poses the following two technical challenges:

1. Maintaining the type environment for non-linear pattern matching (Δ in Fig. 2) in Haskell (Sects. 3 and 4);
2. Implementing matchers and patterns in Haskell (Sect. 3) and achieving ad-hoc polymorphism of patterns (Sect. 5).

The rest of the paper shows the solutions.

3 Typing in Haskell

We start by looking into the type of **matchAll**, which is defined as a function that takes a target whose type is **a**, a matcher whose type is **m**, and a list of match clauses whose type is **[MClause a m b]**:

```
matchAll :: (Matcher m a)
         => a -> m -> [MClause a m b] -> [b]
```

The type constraint **(Matcher m a)** asserts that **m** is a matcher for **a**. **Matcher** is a multi-parameter type class [7] that declares no class method:

```
class Matcher m a
```

We can declare that data of some type are matchers for data of some type by instance declaration for **Matcher**. The following instance declaration asserts that **Eql** is a matcher for the type **a** that is an instance of **Eq**. Basically, a matcher is defined as a singleton type.

```
data Eql = Eql
instance (Eq a) => Matcher Eql a
```

MClause a m b is a type for match clauses as its name implies. The type variables **a**, **m**, and **b** represent the types of the target, matcher, and body expression, respectively. **MClause a m b** is defined using existential types as follows:

```
data MClause a m b = forall vs. (Matcher m a) => MClause (Pattern a m '[] vs) (
    ↪ HList vs -> b)
```

The first argument of **MClause** is a pattern for **a**. This pattern creates a new binding whose type is **vs**. The second argument of **MClause** is the body of the match clause. The body of a match clause is represented as a function that takes a heterogeneous list [10] **HList vs** and returns **b**. The type variable **vs** is existentially quantified, because each pattern of the match clauses in the same pattern-match expression generally makes different bindings. **MClause** may be an illustrative example of existential types. Heterogeneous lists have two data constructors **HNil** and **HCons**.

```
data HList xs where
  HNil :: HList '[]
  HCons :: a -> HList as -> HList (a ': as)
```

The **mc** quasi-quoter transforms the description of a match clause to the expression that is evaluated to a datum of **MClause**. For example, **[mc| $x : $y : _ -> (x, y) |]** is rewritten as follows:

```
MClause (cons (PatVar "x") (cons (PatVar "y") Wildcard))
        (\HCons x (HCons y HNil) -> (x, y))
```

The pattern of this match clause contains two pattern variables **$x** and **$y**. They are transformed to **PatVar "x"** and **PatVar "y"**, respectively. The **PatVar** data constructor takes the variable name as the argument just for improving the readability of the result of transforming a pattern. We implement the internal pattern-match algorithm of the proposed library so that pattern variables bind values in the order of occurrence. The wildcard **_** is rewritten to the data constructor **Wildcard**. The body expression is also transformed into a function that takes a heterogeneous list that consists of values bound to **x** and **y**. The infix pattern constructor "**:**" is desugared to the **cons** pattern constructors.

Next, we look into the type of patterns. The **Pattern** type constructor takes four arguments: **a** is the type of data with which the pattern matches; **m** is the type of a matcher that is used in the pattern-match expression; **ctx** is the type of an intermediate pattern-match result that is a list of data bound in the left-side of the pattern; **vs** is the list of types that are bound to the pattern variables in the pattern. The data type "**Pattern a m ctx vs**" has three data constructors. These three data constructors represent wildcards, pattern variables, and user-defined patterns, respectively.

```
1 data Pattern a m ctx vs where
2   Wildcard :: (Matcher m a)
3            => Pattern a m ctx '[]
4   PatVar   :: (Matcher m a)
5            => String
6            -> Pattern a m ctx '[a]
7   Pattern  :: (Matcher m a)
8            => (HList ctx -> m -> a -> [MList ctx vs])
9            -> Pattern a m ctx vs
```

The type-level empty list **`'[]`** in line 3 represents that a wildcard makes no new bindings. The type-level singleton list **`'[a]`** in line 6 represents that a pattern variable makes a single binding. User-defined patterns are represented using the **`Pattern`** data constructor. The **`Pattern`** data constructor takes a function that defines the pattern-match algorithm. The method for defining pattern-match algorithms will be presented in Sect. 5. No data constructor for value patterns is built-in, except for wildcards and pattern variables. This is because value patterns are defined as a user-defined pattern using **`Pattern`** in miniEgison. **`MList`** is a data type for representing a stack of matching atoms. The details of these definitions will be discussed in subsequent sections. The implementation of value patterns will be discussed in Sect. 5.3.

4 Implementing Pattern-Match Expressions

This section presents the implementation of the internal pattern-match algorithm of the Egison pattern-match system. Section 4.1 gives an overview of the pattern-match algorithm. Sections 4.2 and 4.3 discuss the implementation of the algorithm in Haskell. The internal pattern-match algorithm of Egison is also described in Sects. 5 and 7 of the original paper of Egison [3].

4.1 Overview of Pattern-Match Algorithm

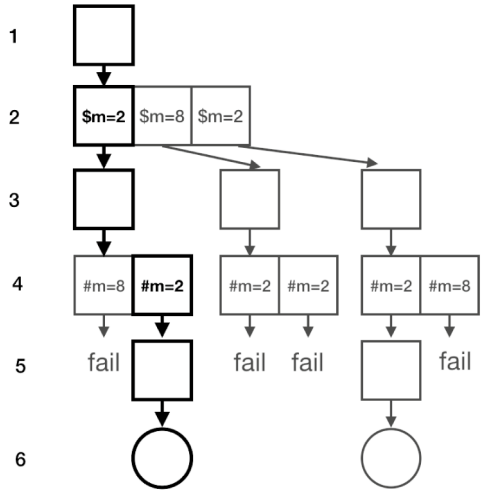
We exemplify the pattern-match algorithm inside Egison by observing what happens in the evaluation of the pattern-match expression in Fig. 3a. Figure 3b, c show the reduction sequence of evaluation that occurs inside miniEgison.

First, the initial *matching state* (the first row in Fig. 3b, c) is created from the pattern, matcher, and target of the **`matchAll`** expression. In Egison, pattern matching is implemented as iterated steps of 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, matcher, and target.

In the next step, three matching states are generated from the initial matching state (the second row in Fig. 3b, c). The reduction from the first row to the second row proceeds by referring to the pattern-match algorithm of the cons pattern for a multiset. In each step of a pattern-match process, the top matching atom in is popped off. From this matching atom, a list of lists of the next matching atoms is generated. Each of the next matching atoms is pushed on the stack of matching atoms of the

```
matchAll [2, 8, 2] (Multiset Eql)
  [[mc| $m : #m : _ -> m|]]
-- [2, 2]
```

(a) Sample pattern-match expression.



(b) Search tree for the pattern-match expression in figure 3a. The rectangles and circles represent the matching states and final pattern-match results, respectively.

1	`([($m : #m : _, Multiset Eql, [2,8,2])], [])`

2	`([($m, Eql, 2), (#m : _, Multiset Eql, [8, 2])], [])`
	`([($m, Eql, 8), (#m : _, Multiset Eql, [2, 2])], [])`
	`([($m, Eql, 2), (#m : _, Multiset Eql, [2, 8])], [])`

3	`([(#m : _, Multiset Eql, [8, 2])], [2])`
4	`([(#m, Eql, 8), (_, Multiset Eql, [2])], [2])` `([(#m, Eql, 2), (_, Multiset Eql, [8])], [2])`
5	`([(_, Multiset Eql, [8])], [2])`
6	`([], [2])`

(c) Reduction path of matching states. The matching state is represented as a pair of a stack of matching atoms and intermediate pattern-match results. The matching states highlighted in this figure correspond to the highlighted rectangles and circles in figure 3b.

Fig. 3 Internal pattern-matching algorithm

matching state. As a result, a single matching state is reduced to multiple matching states in a single reduction step. Pattern matching is recursively executed for each matching state.

The matching state in the third row is generated by reducing the first matching state in the second row. In Fig. 3c, the matching state that is not grayed-out in the i-th row is reduced to matching states in the $(i + 1)$-th row. In this step, the top matching

atom is popped off, and a new binding is added to the intermediate pattern-match result because the pattern of the top matching atom is the pattern variable **$m**.

Pattern matching succeeds and returns the final pattern-match result when a stack becomes empty, as shown in the matching state in the sixth row.

4.2 Typing Matching States

In the implementation of the algorithm in Sect. 4.1, the difficulty arises in typing matching states and matching atoms. This section explains how we define the types of matching states and matching atoms.

4.2.1 Heterogeneous Lists

A pattern-match result is represented as a heterogeneous list [10] in miniEgison because of the following two reasons: (i) the number of the pattern variables that appear in a pattern is not fixed; (ii) the types of objects that are bound to the pattern variables differ for each pattern variable. The implementation of heterogeneous lists in Haskell was first proposed in [10]. Heterogeneous lists are easier to implement with GHC extensions for type-level programming, which have been developed later 8, 21]. The implementation of heterogeneous lists in miniEgison is similar to that of [21].

4.2.2 Matching Atoms

A matching atom is represented with the **MAtom** data constructor that takes three arguments, a pattern, a matcher, and a target. The constraint **(Matcher m a)** asserts that **m** is the type of a matcher for **a**. The types of a target **a** and a matcher **m** are existentially quantified because each matching atom in a stack of matching atoms contains a pattern, matcher, and target for a different type. The **MAtom** type constructor takes only two arguments, **ctx** and **vs** that are also arguments of **Pattern**: the type variable **ctx** represents an intermediate pattern-match result; the type variable **vs** represents a list of types bound to the pattern variables by processing this matching atom.

```
data MAtom ctx vs = forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```

4.2.3 A Stack of Matching Atoms

GADTs and type-level programming play essential roles in representing the type of stacks of matching atoms. **MList** is a data type for representing a stack of matching atoms. The **MList** type constructor takes two arguments. Both arguments are represented using heterogeneous lists [10]. The first argument **ctx** represents an intermediate pattern-match result, the types of values that have already been bound to the pattern variables. The second argument **vs** represents the types of values that are going to be

bound to the pattern variables. The append operator for type-level lists is denoted by "**:++:**".

```
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```

MList has two data constructors. **MNil** represents the empty stack of matching atoms. The second argument of **MNil** is a type-level empty list because the empty stack of matching atoms makes no more new bindings. **MCons** takes two arguments. The first argument is the top matching atom, and the second argument is the rest of the stack.

4.2.4 Matching States

A matching state consists of an intermediate pattern-match result (of type **HList xs**) and a stack of matching atoms (of type **MList xs ys**). The type variable **vs** represents a list of types bound to the pattern variables in the pattern after processing **MState**. The type-level list bound to **vs** should be equal to the concatenation of **xs** and **ys**. The operator "**~**" represents that left-side is equal to right-side.

```
data MState vs where
MState :: vs ~ (xs :++: ys)
       => HList xs
       -> MList xs ys
       -> MState vs
```

4.3 Implementation of the Pattern-Match Algorithm in Haskell

This section implements the pattern-match algorithm discussed in Sect. 4.1 using the definition of matching states in Sect. 4.2. This section shows the implementation of **matchAllDFS**, which is simpler than **matchAll**. From the implementation of **matchAllDFS**, we can see the validity of the type definitions of matching states and matching atoms in Sect. 4.2.

The **matchAllDFS** expression is defined as a function as shown in Fig. 4a. The type of **matchAllDFS** is exactly the same as the type of **matchAll** shown in Sect. 3. The initial matching state is constructed and passed to the **processMStatesAllDFS** function, which takes a list of matching states and returns all the pattern-match results. Each pattern-match result returned by **processMStatesAllDFS** is applied to **f**, the body of the match clause. Then, let us look into the implementation of **processMStatesAllDFS** shown in Fig. 4b. The

```
1 matchAllDFS :: (Matcher m a) => a -> m -> [MClause a m b] -> [b]
2 matchAllDFS tgt m [] = []
3 matchAllDFS tgt m ((MClause pat f):cs) =
4   let results = processMStatesAllDFS [MState HNil (MCons (MAtom pat m tgt) MNil
      ↪ )] in
5   map f results ++ matchAllDFS tgt m cs
```

(a) The `matchAllDFS` function

```
1 processMStatesAllDFS :: [MState vs] -> [HList vs]
2 processMStatesAllDFS [] = []
3 processMStatesAllDFS (MState rs MNil:ms) = rs:(processMStatesAllDFS ms)
4 processMStatesAllDFS (mstate:ms) = processMStatesAllDFS ((processMState mstate)
    ↪  ++ ms)
```

(b) The `processMStatesAllDFS` function

```
1 processMState :: MState vs -> [MState vs]
2 processMState (MState rs (MCons (MAtom pat m tgt) atoms)) =
3   case pat of
4     Wildcard -> [MState rs atoms]
5     PatVar _ -> case patVarProof rs (HCons tgt HNil) atoms of
6                   Refl -> [MState (happend rs (HCons tgt HNil)) atoms]
7     Pattern p -> let matomss = p rs m tgt in
8                  map (\newAtoms -> MState rs (mappend newAtoms atoms)) matomss
```

(c) The `processMState` function

Fig. 4 The implementation of the internal pattern-match algorithm

first clause (line 2) describes the base case. The second clause (line 3) describes the case when the stack of matching atoms is empty, that is, when pattern matching succeeds. The final clause (line 4) calls **`processMState`**, which proceeds the pattern-match process one step further. The **`processMState`** function takes a matching state and returns a list of the next matching states. The **`processMState`** function is defined as shown in Fig. 4c. The first match clause (line 4) handles a wildcard. When the pattern is a wildcard, the top matching atom is just popped out. The second match clause (lines 5–6) handles a pattern variable. The top matching atom is popped out, and a new value is added to the tail of the intermediate pattern-match result. The **`case`** expression for the return value of **`patVarProof`** tells the GHC type checker the list append associativity rule **`((xs :++: '[a]) :++: ys) : : (xs :++: ('[a] :++: ys))`**. The operator "**`: :`**" is called propositional equality and represents that we can prove left-side is equal to right-side. The third match clause (lines 7–8) handles a user-defined pattern. The **`Pattern`** data constructor takes a single argument, which is a function that takes an intermediate pattern-match result, a matcher, and a target as the arguments and returns a list of lists of next matching atoms. Each list of next matching atoms is prepended to the stack of matching atoms. The function **`happend`** (resp., **`mappend`**) concatenates a pair of heterogeneous lists (resp., stacks of matching atoms).

5 Defining Matchers

The customizability of pattern-match algorithms is an important characteristic of Egison. This section shows how we can implement this feature in Haskell.

5.1 Unordered Pairs

First, we start by defining a matcher for unordered pairs. The **UnorderedPair m** matcher is a matcher for a pair of elements whose order is insignificant. The elements of an unordered pair are pattern-matched using **m** as a matcher. For example, **UnorderedPair Eql** can be used as follows:

```
matchAll (1,2) (UnorderedPair Eql)
  [[mc| upair #2 $x -> x |]] -- [1]
```

The **upair** pattern constructor takes two patterns as its arguments. Each of them is pattern-matched with an element of the pair ignoring the order of them. In the above sample, the value pattern **#2** that is the first argument of **upair** matches **2** that is the second element of the target tuple. As a result, **$x** is bound to **1** that is the first element of the target tuple.

The **UnorderedPair m** matcher is defined as a data type whose type and data constructor are identical. The **UnorderedPair m** matcher is an instance of the **Matcher** type class. This instance declaration asserts that **UnorderedPair m** is a matcher for a pair of elements of the type **a** when **m** is a matcher for the type **a**.

```
data UnorderedPair m = UnorderedPair m
instance (Matcher m a)
      => Matcher (UnorderedPair m) (a, a)
```

The **upair** pattern constructor can be defined as a function whose type is defined as follows:

```
1 upair :: (m ~ (UnorderedPair m'), Matcher m (a, a),  Matcher m' a)
2       => Pattern a m' ctx xs
3       -> Pattern a m' (ctx :++: xs) ys
4       -> Pattern (a, a) m ctx (xs :++: ys)
```

The type constraints (line 1) declare that **a** is a type of elements of unordered pairs and **m** and **m′** are matchers for unordered pairs and their elements, respectively. The **upair** pattern constructor takes two patterns for elements for unordered pairs (lines 2–3) and returns a pattern for unordered pair (line 4). The type of a pattern-match result of the first and second argument of **upair** is bound to the type variable **xs** and **ys**, respectively. The pattern-match result **xs** is appended to the intermediate pattern-match result **ctx** in line 3 for handling non-linear patterns.

The **upair** pattern constructor is defined using the **Pattern** data constructor. **Pattern** takes a function that takes an intermediate pattern-match result, a matcher, and a target and returns a list of lists of next matching atoms. Note that the difference between two lists of the next matching atoms returned by **upair** is only the order of **t1** and **t2**. The first list of the next matching atom represents pattern matching for the target pair in order. The second list of the next matching atom represents pattern matching for the target pair in reverse order. The internal pattern-match algorithm pushes each list of matching atoms on the stack of matching atoms.

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
  [MCons (MAtom p1 m' t1) (MCons (MAtom p2 m' t2) MNil)
  ,MCons (MAtom p1 m' t2) (MCons (MAtom p2 m' t1) MNil)])
```

5.2 Matchers with Polymorphic Patterns

Ad hoc polymorphism of patterns is achieved by defining pattern constructors as functions of type classes. This is exemplified through the implementation of the cons pattern for lists and multisets. Likewise the **UnorderedPair m** matcher, **List m** and **Multiset m** are defined as follows:

```
data List m = List m
instance (Matcher m a) => Matcher (List m) [a]

data Multiset m = Multiset m
instance (Matcher m a) => Matcher (Multiset m) [a]
```

The difference from unordered pairs is in the definition of pattern constructors. The pattern constructors for collections are defined as functions of the **CollectionPat** class.[2]

```
class (Matcher m a) => CollectionPat m a where
  cons :: (a ~ [a'], m ~ (f m'))
       => Pattern a' m' ctx xs
       -> Pattern a m (ctx :++: xs) ys
       -> Pattern a m ctx (xs :++: ys)
```

The type equality constraints on **cons** declare that **a** is the type of a list of **a′** and **m** is a matcher whose form is "**_ m′**". These type constraints are necessary to help GHC to infer the types of **a′** and **m′**. The pattern-match algorithm for the cons pattern is bound to **cons**, because we implement the **mc** quasi-quoter to desugar "**:**" in patterns to the **cons** pattern constructors. For example, the pattern **$x : _** is desugared to **cons (PatVar "x") Wildcard**.

[2] The definitions of the **nil** and **join** patterns are omitted.

Then, the pattern-match algorithms for the cons pattern of **`List m`** and **`Multiset m`** can be defined just as an instance of **`CollectionPat`**.

```
instance (Matcher m a)
      => CollectionPat (List m) [a] where
  cons p1 p2 = Pattern (\_ (List m) tgt ->
    case tgt of
      [] -> []
      x:xs ->
        [MCons (MAtom p1 m x) (MCons (MAtom p2 (List m) xs) MNil)])

instance (Matcher m a)
      => CollectionPat (Multiset m) [a] where
  cons p1 p2 = Pattern (\_ (Multiset m) tgt ->
    map (\(x, xs) -> MCons (MAtom p1 m x) (MCons (MAtom p2 (Multiset m) xs)
    ↪ MNil))
        (matchAll tgt (List m) [[mc| $hs ++ $x : $ts -> (x, hs ++ ts) |]]))
```

5.3 Matchers with Value Patterns

This section explains how to define a pattern-match algorithm for value patterns.

A match clause that contains a value pattern such as **`[mc| $x : #(x + 1) : $y : _ -> (x, y) |]`** is transformed as follows:

```
1 MClause (cons (PatVar "x")
2          (cons (valuePat (\HCons x HNil -> x + 1))
```

The **`valuePat`** pattern constructor in line 2 takes a function that takes an intermediate result of pattern matching and returns a value to be compared with the target. Like the **`cons`** pattern constructor, **`valuePat`** is defined as a function of a type class to achieve ad hoc polymorphism.

```
class ValuePat m a where
  valuePat :: (Matcher m a, Eq a)
           => (HList ctx -> a)
           -> Pattern a m ctx '[]
```

Here is a definition of the **`Eql`** matcher. **`Eql`** is a matcher for data types that belong the **`Eq`** type class. **`Eql`** uses **`==`** for pattern-matching value patterns. We have shown several samples that use **`Eql`** in Sect. 1.1. The definition is similar to those of **`List m`** and **`Multiset m`** in Sect. 5.2. The difference is that the intermediate pattern-match result **`ctx`**, which is the first argument of the function passed to

`Pattern`, is passed to **`f`**, the argument of **`valuePat`**. When **`f ctx`** is equal to **`tgt`**, pattern matching succeeds.

```
data Eql = Eql
instance (Eq a) => Matcher Eql a
instance (Eq a) => ValuePat Eql a where
  valuePat f = Pattern (\ctx _ tgt ->
                          [MNil | f ctx == tgt])
```

Ad hoc polymorphism of **`valuePat`** allows us to define equality for non-free data types. For example, we can define a pattern-match method for a value pattern of a multiset as follows:

```
instance (Matcher m a, Eq a, ValuePat m a)
      => ValuePat (Multiset m) [a] where
  valuePat f = Pattern (\ctx (Multiset m) tgt ->
    match (f ctx, tgt) (Pair (List m) (Multiset m))
      [[mc| (nil, nil) -> [MNil] |],
       [mc| ($x : $xs, #x : #xs) -> [MNil] |],
       [mc| _ -> [] |]])
```

6 Differences Between Scheme and Haskell Implementations

The implementation of the proposed library is mostly a direct translation of the Scheme implementation [2]. We can type the program for the internal pattern-match algorithm of the proposed library by utilizing heterogeneous lists, existential types, and GADTs as explained in previous sections.

However, there is a part we modified from the Scheme implementation. That part is the data type definition for patterns and matchers. Patterns are tagged lists and matchers are functions in Scheme, whereas patterns are functions and matchers are data of singleton type in Haskell. We cannot transfer tagged lists in Scheme to algebraic data types in Haskell because Haskell's data family is not allowed to overlap. If we were to define patterns as data in Haskell, we would write a program as follows:

```
data family Pattern a

data instance Pattern a =
  Wildcard
  PatVar String

data instance (Eq a) => Pattern a =
  ValuePat a

data instance Pattern [a] =
  NilPat
  ConsPat (Pattern a) (Pattern [a])
  JoinPat (Pattern [a]) (Pattern [a])
```

However, the above instance declarations are invalid because they are overlapping. For this reason, we implement patterns as functions.

In the Scheme implementation, matchers are functions that take a pattern and a target and return lists of matching atoms. In the functional definition of matcher, we pattern-match a pattern and describe pattern-match algorithms for each pattern in the corresponding match clause. We use a traditional pattern-match facility provided by a Scheme library [20] for pattern matching against patterns. For example, a matcher for unordered pairs is defined as follows:

```
(define UnorderedPair
  (lambda (M)
    (lambda (p t)
      (match p
        ((upair px py)
         (match t
           ((x y) `(((,px ,M ,x) (,px ,M ,x))
                    ((,px ,M ,y) (,px ,M ,x))))))
        (pvar `{{[,pvar Something ,t]}})))))
```

The original Egison programming language treats matchers as special objects and provide a special syntax for defining matchers, but the matcher definition in Egison is very similar to Scheme.

```
unorderedPair m :=
  matcher
    | ($, $) as (m, m) with
      | ($x, $y) -> [(x, y), (y, x)]
    | $ as (eq) with
      | $tgt -> [tgt]
```

This pattern matching for patterns in matcher definitions is sometimes important for improving the execution performance [4]. For example, let us consider the pattern-match expression that enumerates pairs of prime numbers whose form is $(p, p + 6)$:

```
take 5 (matchAll primes (List Eql)
  [[mc| _ ++ $p : (_ ++  #(p + 6) : _) -> (p, p + 6) |]])
-- [(5,11),(7,13),(11,17),(13,19),(17,23)]
```

The above program is very slow, because it enumerates all combinations of prime numbers like (3, 5), (3, 7), (3, 11), (3, 13), (3, 17), and so on. We need not enumerate the pairs that follow (3, 11), if we assume the prime numbers are lined up in ascending order. To avoid this unnecessary enumeration, we introduce a new matcher specialized to sorted lists by defining a pattern-match algorithm for nested constructor patterns whose form is "$\_ + + \#\mathbf{v} : : \_$". In the Scheme implementation, the matchers defined by functions cannot define pattern-match algorithms for such nested constructor patterns.

7 Integration with Existing Haskell Pattern Matching

This section introduces the features of miniEgison for improving the compatibility with the existing Haskell pattern-match facility.

7.1 View Interface

This section introduces the view interface, a handy interface to replace the **match** expression. The view interface allows us to use miniEgison patterns in binding positions of function arguments. The view interface increases the familiarity of miniEgison because Haskell users often pattern-match function arguments directly without using the **case** expression, which is analogous to the **match** and **matchAll** expressions. For example, we can define **delete** using the view interface as follows:

```
1 delete :: Eq a =>  a -> [a] -> [a]
2 delete x ([view| $hs ++ #x : $ts as List Eql |]) =
3   hs ++ ts
4 delete _ xs = xs
```

We provide the **view** quasi-quoter whose form is **[view|** *p* **as** *m* **|]**. The **view** quasi-quoter is expanded to a Haskell pattern that uses view patterns [1]. For example, lines 2–3 of the above program is expanded as follows:

```
1 delete x ((\x_a5aP ->
2              matchAll x_a5aP (List Eql)
3                [[mc| $hs ++ #x : $ts -> (hs, ts) |]])
4           -> (hs, ts) : _) =
5   hs ++ ts
```

The **view** quasi-quoter is expanded to the **matchAll** expression whose pattern and matcher are specified in the **view** interface. The variable **x_a5aP** is auto-generated by Template Haskell. This **matchAll** expression returns a list of values bound to the pattern variables in this **matchAll** expression (**$hs** and **$ts** in line 3). Using the function of view patterns, the first **matchAll** result is bound to Haskell variables **hs** and **ts** in line 4. View patterns also allows the value bound to **x**, the first argument of delete, to be referred to from within the view interface, the second argument of **delete**.

The view interface is a good alternative to the **match** expression but it cannot substitute **matchAll** expressions entirely. This is because Haskell patterns can handle only a single pattern-match result. Therefore the view interface inside a Haskell pattern is also subject to the same restriction.

7.2 Matchers for Algebraic Data Types

We provide a simple interface for generating matchers for algebraic data types. Pattern matching for algebraic data types is necessary for many situations. However, defining matchers for algebraic data types in the manner explained in Sect. 5 is a very tedious task. Therefore, a simple interface for defining matchers for algebraic data types is preferable. We provide a Template Haskell function **`makeMatcher`** for this purpose. Using **`makeMatcher`**, we can define the **`CardM`** matcher used in Fig. 1 as follows:

```
data Card = Card (Suit :%: Eql) (Integer :%: Eql)
makeMatcher ''Card
```

The double quote in the argument of **`makeMatcher`** is a special notation from Template Haskell to specify a type name. The infix operator "**`:%:`**" is implemented as a type family that immediately returns the type of the left-hand side. The datatype definition for **`Card`** is thus equivalent to that of Fig. 1. As **`makeMatcher`** syntactically analyzes the definition of the data type, it is able to interpret "**`:%`**" to specify the matcher for the fields. The last line of the above program is expanded to the program that is equivalent to the following program:

```
data CardM = CardM
instance Matcher CardM Card

card :: Pattern Suit Eql ctx xs
     -> Pattern Integer Eql (ctx :++: xs) ys
     -> Pattern Card CardM ctx (xs :++: ys)
card p1 p2
  = Pattern
      (\ _ CardM (Card x1 x2)
        -> [MCons (MAtom p1 Eql x1)
              ((MCons (MAtom p2 Eql x2)) MNil)]
      )
```

8 Performance

The section discusses the performance of the proposed library. First, we compare the performance of programs that enumerate two permutations of elements, namely **`perm2`**, implemented in a traditional functional style in Haskell and pattern-match-oriented style in Haskell, Scheme, and Egison. For the Scheme implementation, we used the Scheme macro library that implements Egison pattern-match facility [2]. Table 1 shows the benchmark results. We implemented the benchmark program **`perm2`** in different implementation languages as follows.

Table 1 Execution time of **perm2** programs in milliseconds

perm2	$n = 50$	$n = 100$	$n = 200$	$n = 400$	$n = 800$	$n = 1600$
Functional style	7.82	6.86	10.2	15.9	36.9	74.3
miniEgison (Version 1.0.0)	8.89	10.6	17.7	39.1	84.9	308
Scheme (egison-scheme 1.0.0)	132	170	429	1190	5100	16,000
Egison (Version 3.10.3)	841	1540	4170	15,000	58200	n/a

Listing 1: Benchmark: Functional Style in Haskell

```
perm2 :: Int -> [(Int, Int)]
perm2 n = go [1 .. n] []
 where
  go [] _ = []
  go (x : xs) rest =
    [ (x, y) | y <- rest ++ xs ] ++ go xs (rest ++ [x])
```

Listing 2: Benchmark: Pattern-Match-Oriented Style in Haskell

```
perm2 :: Int -> [(Int, Int)]
perm2 n =
  matchAllDFS [1..n] (Multiset Something)
    [[mc| $x : $y : _ -> (x, y) |]]
```

Listing 3: Benchmark: Pattern-Match-Oriented Style in Scheme

```
(define (perm2 n)
  (match-all (iota n 1) (Multiset Something)
    ((cons x (cons y _)) `(,x, y))))
```

Listing 4: Benchmark: Pattern-Match-Oriented Style in Egison

```
perm2 n :=
  matchAllDFS [1..n] as multiset something with
    $x :: $y :: _ -> (x, y)
```

Table 2 Execution time of **`pokerHand`** programs in milliseconds

pokerHand	$n = 800$	$n = 1600$	$n = 3200$	$n = 6400$	$n = 12,800$
Functional style (general-games 1.1.1)	162.6	317.9	597.5	1228	2336
miniEgison (Version 1.0.0)	319.9	628.9	1204	2410	4693
Scheme (egison-scheme 1.0.0)	3130	6020	12520	n/a	n/a
Egison (Version 3.10.3)	40630	n/a	n/a	n/a	n/a

We also benchmarked programs that analyze a poker hand of randomly selected n hands. Table 2 shows the benchmark results. We use the program in general-games package [6] which is relatively popular on Hackage as a functional poker hand analyzer. The poker hand analyzer in pattern-match-oriented style is shown in figure 1. The complete benchmarking program is accessible on GitHub [12].

We compiled our benchmarks by the Glasgow Haskell Compiler (GHC) 8.8.2 and ran them on a 3.2 GHz octa core AMD Ryzen 7 2700 processor with 32GB of RAM. We passed the options **`-O3 -threaded -rtsopts -with-rtsopts=-N`** to GHC for compiling the above programs. We used Gauche Scheme interpreter 0.9.9 for the Scheme benchmarks. The execution time is estimated using an ordinary least-squares regression model. We used the criterion [13] benchmarking library to get these results. The benchmark results show that the pattern-match-oriented program written in miniEgison runs much faster than the original Egison interpreter and the Scheme library, but slower than the functional style program. The performance difference from the functional style program comes from our implementation of the internal pattern-match algorithm as a reduction of matching states, which makes it hard to be optimized by GHC. We are seeking a more efficient compilation method of Egison pattern matching as important future work. Currently, we are developing a method for compiling Egison pattern matching to a Haskell program that uses a backtracking monad [11].

9 Conclusion

This paper presented a Haskell library that implements the user-customizable non-linear pattern matching with backtracking which has been implemented in the Egison programming language. We showed that Egison pattern matching can be statically typed. Thanks to the static type system, we can detect type errors at compile time before executing programs. For this purpose, we proposed a new set of typing rules and a method for embedding these typing rules in Haskell utilizing GHC extensions. These typing rules allow us to handle ad-hoc polymorphism of patterns and non-linear patterns. We utilized several GHC extensions for embedding these typing rules into Haskell. The proposed library is an illustrative application of the following GHC extensions:

1. Template Haskell desugars match clauses and allows an intuitive representation of non-linear patterns (Sect. 3);
2. Datatype promotion is used to represent heterogeneous lists. Heterogeneous lists are used to represent pattern-match results (Sects. 3 and 4);
3. GADTs are used for typing non-linear patterns (Sect. 3) and a stack of matching atoms (Sect. 4);
4. Existential types are used for typing match clauses (Sect. 3) and a stack of matching atoms (Sect. 4);
5. Multi-parameter type classes are used to represent the relationships between a target and a matcher (Sect. 3) and achieve ad-hoc polymorphism of patterns (Sect. 5);
6. View patterns for using the proposed pattern-match facility in top-level definitions (Sect. 7).

The proposed library makes the Egison pattern matching facility accessible from Haskell users—a majority of functional programmers. Not only that, the proposed library is much faster than the original Egison interpreter and endures practical use. We have already utilized miniEgison for implementing the computer algebra system implemented in the Egison interpreter. We hope the proposed library increases the number of Egison fans and leads to more inventions of practical applications of pattern-match-oriented programming.

Acknowledgements We thank Yuichi Nishiwaki for constructive discussion while implementing the proposed library.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. ViewPatterns-GHC. https://ghc.haskell.org/trac/ghc/wiki/ViewPatterns (2015). [Online; accessed 14-June-2018]
2. Egi, S.: Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types. In: The Scheme and Functional Programming Workshop (2019)
3. Egi, S., Nishiwaki, Y.: Non-linear pattern matching with backtracking for non-free data types. In: Asian Symposium on Programming Languages and Systems , Springer, New York, pp. 3–23 (2018)
4. Egi, S., Nishiwaki, Y.: Functional Programming in Pattern-Match-Oriented Programming Style. In: The Art, Science, and Engineering of Programming. Vol. 4, Issue 3, Article 7 (2020)
5. Erwig, M.: Active patterns. Implementation of Functional Languages (1996)
6. Gorski, C.A.: General-games: Library supporting simulation of a number of games. https://hackage.haskell.org/package/general-games (2018). [Online; accessed 10-May-2020]
7. Hudak, P., Hughes, J., Jones, S. P., Wadler, P.: 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, pp. 1–55 (2007)
8. Jones, S. P., Washburn, G., Weirich, S.: Wobbly Types: Type Inference for Generalised Algebraic Data Types. Tech. rep., Technical Report MS-CIS-05-26, Univ. of Pennsylvania (2004)

9. Kawata, A.: egison/typed-egison: Typed Egison. https://github.com/egison/typed-egison, 2018. [Online; accessed 10-Oct-2019]
10. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell* (2004), ACM, pp. 96–107
11. Kiselyov, O., Shan, C., Friedman, D. P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005* (2005), pp. 192–203
12. Kori, M., Egi, S.: egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. https://github.com/egison/egison-haskell, 2019. [Online; accessed 21-Feb-2019]
13. OSullivan, B.: criterion: a Haskell microbenchmarking library. http://www.serpentine.com/criterion/, 2014. [Online; accessed 13-Mar-2020]
14. Pope, R., Yorgey, B.: first-class-patterns: First class patterns and pattern matching, using type families. https://hackage.haskell.org/package/first-class-patterns, 2019. [Online; accessed 13-Mar-2020]
15. Sheard, T., Jones, S. P.: Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (2002), ACM, pp. 1–16
16. Team, G.: 13.1. Language options — Glasgow Haskell Compiler 8.8.1 User's Guide. https://downloads.haskell.org/~ghc/8.8.1/docs/html/users_guide/glasgow_exts.html, 2019. [Online; accessed 18-Dec-2019]
17. Tullsen, M.: First class patterns. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings* (2000), E. Pontelli and V. S. Costa, Eds., vol. 1753 of *Lecture Notes in Computer Science*, Springer, pp. 1–15
18. Turner, D.: Some history of functional programming languages. In *International Symposium on Trends in Functional Programming* (2012), Springer, pp. 1–20
19. Wadler, P.: 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)
20. Wright, A. K., and Duba, B. F.: Pattern matching for Scheme. *Unpublished manuscript* (1993)
21. Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J. P.: Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation* (2012), ACM, pp. 53–66