Pattern-Matching-Oriented Programming Language and Computer Algebra System as Its Application
（パターンマッチ指向プログラミング言語とその応用としての数式処理システム）

by

Satoshi Egi
江木聡志

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 3, 2021
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Masami Hagiya　萩谷昌己
Professor of Computer Science

ABSTRACT

A new programming language that widens the range of algorithms that we can concisely describe is important not only because it makes programming of the existing problems easy but also because it widens the scope of computer science by enabling us to deal with problems that we have avoided because programming has been difficult. In this thesis, we propose the Egison programming language with two independent new features: (i) a pattern-match facility for non-free data types; (ii) a facility for describing tensor calculus in differential geometry in a form similar to mathematical formulae using index notation.

Pattern matching of Egison features user-customizable non-linear pattern matching with backtracking. Several non-standard pattern constructs derive from this pattern-match facility: loop patterns and indexed pattern variables for representing the repetitions in a pattern; sequential patterns for controlling the order of pattern matching. In the first half of the thesis, we discuss the design and implementation of this pattern-match facility, classify programming techniques utilizing this pattern-match facility, and advocates a new paradigm, called pattern-match-oriented programming. Then, we show three applications of Egison pattern matching. First, we reimplement the basic list processing functions such as delete, concat, and unique and a SAT solver. We can simplify their implementations by confining explicit recursions for backtracking inside an intuitive pattern. Second, we show that Egison pattern matching provides a unified query language that can be used for various kinds of databases. Egison pattern matching allows us to describe non-linear patterns for a set, tree, and graph that are executable with a backtracking algorithm. Third, we show that Egison pattern matching makes the implementation of a computer algebra system easy because it makes the implementation of a pattern-match engine for mathematical expressions compact. Egison pattern matching has two implementations: an interpreter in Haskell and a library implemented using Haskell meta-programming. This Haskell library converts Egison pattern matching to a Haskell program that uses a backtracking monad. We can execute Egison pattern matching as fast as Haskell because we convert Egison pattern matching to an equivalent Haskell program. A computer algebra system is implemented in the Egison interpreter using this Haskell library.

Tensor index notation that makes descriptions of tensor calculus in differential geometry is implemented in this computer algebra system. In the second half of the thesis, we propose a method for importing tensor index notation into programming. First, we propose a set of symbolic index reduction rules that allows us to define tensor operators in differential geometry, such as tensor addition and multiplication, partial derivative, and covariant derivative, without describing index rules that differ for each operator. In addition, we propose a set of index completion rules that allow us to concisely define operators for differential forms, such as wedge product, exterior derivative, and Hodge operator. A differential form is an important concept that allows us to describe formulae in differential geometry in a coordinate independent way. It is known that a differential form can be represented as an anti-symmetric tensor and expressed by omitting its indices in mathematical formulae. If we express differential forms as anti-symmetric tensors with omitted indices in programs as we do in mathematical formulae, we can design a set of index completion rules that conforms to the above index reduction rules. Using this method, we can define operators for differential forms using tensor operators and index notation. It makes definitions of operators for differential forms concise. In addition, the users of this computer algebra system can translate mathematical formulae that contain both tensors and differential forms to programs because both tensor and differential forms are represented as tensors and follow the same index rules. Programs of this computer algebra system are close to a form of formulae in differential geometry. Therefore, it is easy to use this computer algebra system even for researchers of mathematics who are not used to programming.

Finally, we discuss a general method for creating a new language facility by reviewing the processes of designing these two language facilities. To this end, we divide the process of creating a new language facility into three steps, classify our contributions for the two language facilities into these three steps, and list commonalities.

論文要旨

簡潔に記述できるアルゴリズムの範囲を広げる新しいプログラミング言語は，既知のコンピュータ科学の問題のプログラミングを簡単にするだけでなく，従来プログラミングが難しいために敬遠されていた問題を扱いやすくし，コンピュータ科学の扱う範囲を広げるために重要である．本論文は，(i) 集合や，グラフ，数式のように 1 つの定まった標準形を持たないデータ型に対しても適用可能なパターンマッチと (ii) テンソルの添字記法を使って微分幾何の計算を数式に近い形で記述する機能の 2 つの独立した新しい言語機能をもつプログラミング言語 Egison を提案する．

Egison のパターンマッチは，ユーザーがパターンマッチ・アルゴリズムをカスタマイズ可能でかつ，バックトラッキング・アルゴリズムによって実行可能な非線形パターンをサポートするという特徴をもつ．このパターンマッチからは，パターン中の繰り返しを表現するためのループ・パターンと添字付きパターン変数や，パターンマッチの順序を制御するためのシーケンシャル・パターンという新しい種類のパターンが生まれる．本論文の前半では，このパターンマッチ機能をプログラミング言語に実装する手法を述べたあと，このパターンマッチを活かしたプログラミングテクニックをまとめ，パターンマッチ指向プログラミングというパラダイムを提唱する．そして，このパターンマッチの 3 つの応用事例を通してその有効性を実証する．1 つめに，従来は再帰を使って記述されていたバックトラッキングのためのループをパターンの中に押し込めることにより，delete や concat，unique のような基本的なリスト関数と SAT ソルバーを再実装する．2 つめに，集合・ツリー・グラフに対してバックトラッキング・アルゴリズムで実行可能な非線形パターンを記述できるために，さまざまな種類のデータベースに対して統一的な形のクエリー言語を提供できることをみせる．3 つめに，数式に対するパターンマッチ・エンジンをコンパクトに実装できるため，数式処理システムの実装が簡単になることをみせる．Egison のパターンマッチの実装には，Haskell によるインタプリタと，Haskell のメタプログラミング機能を用いて実装されたライブラリがある．Haskell によるライブラリ実装は，Egison パターンマッチをバックトラッキング・モナドを使った Haskell プログラムに変換する．同じ意味の Haskell プログラムに変換するため，Egison パターンマッチが Haskell と同等の速度で実行できる．この Haskell ライブラリを利用して，Egison インタプリタには数式処理システムとしての機能が実装されている．

微分幾何に現れるテンソル計算の記述を簡潔にする添字記法は，この数式処理システムに実装されている．本論文の後半では，このテンソル計算を簡潔に表現するための手法を提案する．まず，微分幾何に現れる演算子（テンソルの足し算や掛け算，偏微分，共変微分など）の定義を簡潔にする添字の簡約ルールを提案する．添字の簡約ルールを適切に設計すると，演算子ごとに異なる添字のルールを記述することなしに微分幾何の演算子を定義できる．さらに，微分形式に対する演算子（ウェッジ積，外微分，ホッジ作用素など）の定義を簡潔にする添字の補完ルールを提案する．微分形式は，座標系の選択によらない微分幾何の数式の表現を可能にする重要な概念である．微分形式は歪対称テンソルとして表現できることが知られ，その添字を省略することによって数式の上で表現されることがある．これと同様に，添字が省略された歪対称テンソルとして微分形式をプログラム上で表現すると，上記のテンソルの添字の簡約ルールと適合する添字の補完ルールを設計できる．この手法を使うと，微分形式の演算子の定義にテンソルの演算子と添字記法が使えるため，

それらの定義が簡潔になる．また，テンソルと微分形式の両方がテンソルとして表現され同一の添字の簡約ルールに従うため，この数式処理システムのユーザーは，テンソルと微分形式が混ざった数式もそのままプログラムに落とすことができる．この数式処理システムのプログラムは微分幾何に現れる数式に近いため，あまりプログラミングに慣れていない数学の研究者にとってもこの数式処理システムは扱いやすく，実際の微分幾何の研究で現れる複雑な計量をもつ複素多様体の不変量の計算にこの数式処理システムは既に使われている．

最後に，これら 2 つの言語機能を設計した過程を振り返ることにより，新しい言語機能を作成するための一般的な手法について論じる．そのために，新しい言語機能作成の過程を 3 つのステップにわけ，本論文で提案した 2 つの言語機能の作成のための貢献をこの 3 つのステップごとに分類し，共通点を列挙する．

Acknowledgements

First of all, I thank my supervisor Masami Hagiya, who provided me with helpful advice and opportunities to meet and discuss with researchers outside the university.

I thank Yoichi Hirai, Ibuki Kawamata, and Kentaro Honda, Hagiya laboratory members when I was a graduate student in the Master's course. They are the first real users of our language and gave me a lot of constructive feedback.

I thank Yuichi Nishiwaki, who is a current Hagiya laboratory member. Thanks to collaboration with him, I succeeded in publishing the first paper on the pattern-match system of our language.

I thank Ryo Tanaka, Takahisa Watanabe, Takuya Kuwahara, Yasuhiro Yamada, Mayuko Kori, Akira Kawata, Momoko Hattori, and Hiromi Ogawa for their important contributions to develop our language.

I thank Hiroki Fukagawa, Yutaka Shikano, Takayuki Muranushi, Naoya Umezaki, Yoshiaki Maeda, and Steven Rosenberg for intensive discussions about differential geometry and physics. These discussions motivated me to import the advanced notations from differential geometry into programming languages.

I thank Michal J. Gajda, Hiromi Hirano, Matthew Roberts, Kimio Kuramitsu, Hidehiko Masuhara, Yasunori Harada, and Ikuo Takeuchi for their helpful feedback on our language and papers.

Finally, I thank my colleagues and supervisors in Rakuten, Inc. I especially thank Yukihiro Matsumoto, Masaya Mori, Will Archer Arentz, Neil Primozich, Pierre Imai, Yusaku Kaneta, Yu Hirate, Takuya Kitagawa, and Hiroshi Mikitani. Thanks to their encouragement and support, I can continue my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The advance of science is inextricably linked with the evolution of notation. New scientific knowledge leads to the invention of new notations, and new notations accelerate the advance of science. The evolution of numeric notation is an obvious example. The numeric notation has been evolved utilizing our knowledge of arithmetic. The evolved numeric notation (e.g., decimal notation) allows us to identify numbers with less effort and plays an important role in deepening our knowledge. New scientific knowledge does not only evolve the existing notation but also expand the target area of notation. The invention of a programming language, a set of notations for describing algorithms, is an example. The invention of computers leads us to use programming languages for describing algorithms instead of natural languages. From the latter half of the twentieth century, many notations have been invented in the field of programming languages. Lexical scoping [20], high-order functions [63, 52], and pattern matching [24] are features specific to programming invented for describing algorithms.

In this thesis, we aim to develop a programming language that expands the range of algorithms that we can describe without translating our recognition of the algorithms for computers. We propose the Egison programming language with two independent new features: (i) a pattern-match facility for *non-free data types*; (ii) a facility for describing tensor calculus in differential geometry in a form similar to mathematical formulae using index notation. The proposed language has already been implemented and is open-sourced.

Non-free data types are data types whose data have no canonical forms. For example, multisets are non-free data types because the multiset $\{a, b, b\}$ has two other equivalent but literally different forms $\{b, a, b\}$ and $\{b, b, a\}$. We developed the pattern-match facility for non-free data types. Pattern matching for poker hands is a simple example of our pattern matching. In Figure 1.1, all poker hands are described in a single pattern. When designing the syntax of our language, we aimed to make this poker hand program as concise as possible. This concise definition of poker hands is achieved by three features of our pattern match facility. First, in line 2 we specify that we pattern-match a list of playing cards as a multiset of cards. For determining poker hand, we do not care about the order of the cards. By pattern matching as a multiset, we can write a pattern without taking care of the order of the cards. In our language, the pattern-match method for multisets is not built-in. The users can define pattern-match methods for each non-free data type including multisets in our language. Second, we use *non-linear patterns* for describing each poker hand. Non-linear patterns allow multiple occurrences of the same variables in a pattern. For example, we use

```
def poker cs :=
  match cs as multiset card with
  | [card $s $n, card #s #(n-1), card #s #(n-2), card #s #(n-3), card #s #(n-4)]
    -> "Straight flush"
  | [card _ $n, card _ #n, card _ #n, card _ #n, _]
    -> "Four of a kind"
  | [card _ $m, card _ #m, card _ #m, card _ $n, card _ #n]
    -> "Full house"
  | [card $s _, card #s _, card #s _, card #s _, card #s _]
    -> "Flush"
  | [card _ $n, card _ #(n-1), card _ #(n-2), card _ #(n-3), card _ #(n-4)]
    -> "Straight"
  | [card _ $n, card _ #n, card _ #n, _, _]
    -> "Three of a kind"
  | [card _ $m, card _ #m, card _ $n, card _ #n, _]
    -> "Two pair"
  | [card _ $n, card _ #n, _, _, _]
    -> "One pair"
  | [_, _, _, _, _] -> "Nothing"
```

Figure 1.1: Pattern matching for poker hand

non-linear patterns for describing that all cards have the same suit (the value pattern `#s` matches the value bound to the pattern variable `$s`). For handling non-linear patterns, we need to traverse the search trees for pattern matching efficiently because non-free data have multiple decompositions. Our main technical challenge is to achieve both the customizability of pattern-match methods and the expressive patterns simultaneously. In this thesis, we present how we designed such a language.

By importing the mathematical notations in tensor calculus into programming, we aim to make programs for tensor calculus concise. Figure 1.2a shows the samples of formulae in tensor calculus. In Figure 1.2b, we express these formulae in our language. We can observe that each term in the formulae is directly translated into our language. Figure 1.2c shows the definition of tensor operators that appear in Figure 1.2b. Each tensor operator is defined in one line. Our main achievement in tensor calculus is that we enable the direct translation of a formula that uses tensor index notation, keeping the definition of tensor operators concise. For this purpose, we have reorganized the mathematical definition of tensor notations because the semantics of some notations are vague and complex to implement them as a part of programming languages.

There is great significance in the study of notation itself. There are several interesting problems: "Is there a general way to find a new notation?", "Is there a formal method for measuring the superiority of multiple notations?" Getting close to the answers to these questions is the underlying motivation of this work.

1.2 Pattern-Match-Oriented Programming

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:

$$R^i{}_{jkl} = \frac{\partial \Gamma^i{}_{jl}}{\partial x^k} - \frac{\partial \Gamma^i{}_{jk}}{\partial x^l} + \Gamma^m_{jl}\Gamma^i{}_{mk} - \Gamma^m_{jk}\Gamma^i{}_{ml}$$

$$\Omega^i{}_j = d\omega^i{}_j + \omega^i{}_k \wedge \omega^k_j$$

(a) Mathematical formulae in tensor calculus

```
def R~i_j_k_l := withSymbols [m]
  ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l

def Ω~i_j := withSymbols [k]
  antisymmetrize (d ω~i_j + ω~i_k ∧ ω~k_j)
```

(b) The above formulae in our language

```
def (.) %t1 %t2 := contractWith (+) (t1 * t2)
def d %A := !(flip ∂/∂) coord A
def (∧) %A %B := A !. B
```

(c) Definition of tensor operators in our language

Figure 1.2: Tensor calculus in our language

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.

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

Interestingly, this basic definition of `map` has been almost unchanged for 60 years since McCarthy first presented the definition of `maplist` in [62]. 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[car[x]]; maplist[cdr[x]; f]]]
```

Recursion used in the above definitions is a mathematically simple but powerful framework for representing computations and has been a very basic construct of functional programming for describing loops in programs. Recursion is heavily used for definitions of many basic functions such as `filter`, `concat`, and `unique` and most of them are also simple enough.

However, as mentioned earlier, there *does* exist a substantial cognitive distance between the recursive definition and the simplest explanation of that definition. To illustrate this gap, we define `mapWithBothSides`, a variation of `map`. We often meet a situation to define a variant of basic functions specific to a target algorithm. Defining these utility functions is one of the cumbersome tasks in programming. Therefore, considering a comfortable method for defining these utility functions is important.

`mapWithBothSides` takes a function of three arguments and a list, and returns a list of applying the function for all three tuples consisting of an initial prefix, the next element, and the remaining suffix. `mapWithBothSides` is used for generating lists by rewriting the element of an input list. This function is useful for handling logical formulae, for example. We define `mapWithBothSides` with a helper function 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 explanation of `map` given at the beginning and the above explanation of `mapWithBothSides` are similar, 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 differences is the lack of a pattern like `hs ++ ts` that divides a target list into an initial prefix and the remaining suffix. For example, the list `[1,2]` has multiple decompositions for the pattern `hs ++ ts`: `[] ++ [1,2]`, `[1] ++ [2]`, and `[1,2] ++ []`. We call this pattern a *join pattern*. Unlike traditional pattern matching for algebraic data types, this pattern has multiple decompositions. Pattern matching that can handle multiple results is necessary for handling join patterns.

In our language Egison whose distinguishing feature is non-linear pattern matching with backtracking [38], we can define `map` and `mapWithBothSides` concisely and in a very similar way. In fact, the author got an idea of the language when he implemented `mapWithBothSides` for implementing an automated theorem conjecturer.

```
def map f xs := matchAll xs as list something with _ ++ $x :: _ -> f x
def mapWithBothSides f xs := matchAll xs as list something with $hs ++ $x :: $ts
    -> f hs x ts
```

In the above program, the join pattern is effectively used. `matchAll` is a key built-in syntactic construct of Egison for handling multiple pattern-matching results. `matchAll` collects all the pattern-matching results and returns a collection where the body expression has been evaluated for each result. `matchAll` takes one additional argument *matcher* that is `list something` in the above cases. 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. `list` is a user-defined function that takes a matcher for the elements and returns a matcher for lists. `list` defines a method for interpreting the cons (`::`) and join (`++`) pattern. `something` is the only built-in matcher in Egison. `something` can be used for pattern-matching arbitrary objects but can handle only pattern variables and wildcards. As a result, `list something` is evaluated as a matcher for pattern-matching a list of arbitrary objects. `_` that appears in a pattern is a wildcard. Pattern variables are prepended with `$`.

These definitions of the variations of `map` are close to the explanation of `map` given at the beginning. We achieved this by hiding the recursions in the definition of `list`, which defines the pattern-matching algorithm for the join patterns. We call this programming style that replaces explicit recursions with intuitive patterns, *pattern-match-oriented programming style*.

In this thesis, we advocate pattern-match-oriented programming as a new programming paradigm. Pattern-match-oriented programming is a paradigm that

makes descriptions of algorithms concise by replacing loops for traversal and enumeration that can be done by simple backtracking. These loops are mixed in programs with loops that are essential for lowering the time complexity of algorithms and make programs complicated. Pattern-match-oriented programming separates these two kinds of loops and allows programmers to concentrate on describing the essential part of algorithms.

The above examples show just a part of the expressiveness of pattern-match-oriented programming. For example, non-linear patterns enable to describe more popular list processing functions such as `unique` and `intersect` in pattern-match-oriented style. Furthermore, pattern matching for general non-free data types (e.g., multisets, sets, and mathematical expressions) diversifies the applications of pattern-match-oriented programming significantly. This thesis introduces full features of the Egison pattern-match-oriented programming language and presents all the techniques we discovered so far for replacing explicit recursions with an intuitive pattern.

1.3 Importing Mathematical Notations from Tensor Calculus

Programming languages are evolved also by importing notations from mathematics. For example, decimal number system, function modularization, and infix notation for basic arithmetic operators have been imported in most programming languages [19]. Importing mathematical notations into programming is sometimes difficult. This is because the semantics of some mathematical notations are vague and complex to implement as a part of programming languages. For example, tensor index notation invented by Ricci and Levi-Civita [71] for dealing with high-order tensors is one such notation.

The latter part of this thesis discusses a method for importing tensor index notation into programming languages. Tensor calculus often appears in the various fields of computer science. Tensor calculus is an important application of symbolic computation [57]. Tensor calculus is heavily used in computational physics [49] and computer visions [47]. Tensor calculus also appears in machine learning to handle multidimensional data [50]. Importing tensor index notation makes programming in these fields easy.

Importing tensor index notation into programming languages is difficult because the symbolic index rules vary by operators, especially by tensor addition and multiplication as shown in Table 1.1. We will show examples. Let u and v be vectors:

- the expression $u_i + v_i$ returns a vector, but $u_i v_i$ is an invalid expression; (The expression $u_i v_i$ returns the inner product in Euclidean spaces where we can switch a superscript and a subscript freely. But we do not take this exception into account.)
- the expression $u^i + v_i$ is an invalid expression, but $u^i v_i$ is valid and returns the inner product;
- the expression $u_i + v_j$ is an invalid expression, but $u_i v_j$ is valid and returns the tensor product.

Thus, each tensor operator has different symbolic index rules. Therefore, we need to specify symbolic index rules for each function when defining them. It is a verbose task. Furthermore, we need to design a syntactic construct for describing index rules. Such a new syntactic construct makes a programming language complex.

Tensor addition	Tensor multiplication
$u_i + v_i$ returns a vector	$u_i v_i$ is invalid
$u_i + v_j$ is invalid	$u_i v_j$ returns a matrix
$u^i + v_i$ is invalid	$u^i v_i$ returns a scalar

Table 1.1: Index rules in mathematics

Tensor addition	Tensor multiplication
$u_i + v_i$ returns a vector	$u_i v_i$ is invalid
$u_i + v_j$ returns a matrix	$u_i v_j$ returns a matrix
$u^i + v_i$ returns a matrix	$u^i v_i$ returns a scalar

Table 1.2: Our simplified index rules

We propose simplified symbolic index rules shown in Table 1.2 to avoid this problem. Our simplified rules keep all the mathematically valid expressions valid. However, some mathematically invalid expressions become valid. For example, $v_i + v_j$ is an invalid expression in mathematics but a valid expression in our rules. Figure 1.3a shows our simplified index rules for tensor addition. By relaxing the rules that way, we can regard many tensor operators as variations of tensor addition. For example, tensor multiplication is regarded as an operation that has an additional procedure, contraction, after multiplying each component of tensors in the same way as tensor addition (figure 1.3b). Contraction is an operation to sum up the diagonal components of a tensor when the tensor has a superscript and subscript with an identical symbol. This point of view reduces the descriptions of symbolic index rules for defining tensor operators. We show that various operators in differential geometry can be defined concisely.

Furthermore, we propose index completion rules for omitted tensor indices. We show that our index completion rules enable us to define operators for differential forms such as the wedge product, exterior derivative, and Hodge star operator.

In our language, we can describe a program whose form is close to formulae in differential geometry. Therefore, it is easy to use this computer algebra system even for researchers of mathematics who are not used to programming. Due to this advantage, our language already has been applied in the research of mathematics [37].

1.4 Outline

The rest of thesis is organized as follows. Chapter 2 proposes a criteria that should be fulfilled for practical pattern matching for non-free data types and designs our programming language that satisfies the criteria. Chapter 3 discusses how programming changes by utilizing our pattern-match system and proposes a new programming paradigm called pattern-match-oriented programming. Chapter 4 presents our Haskell library that embeds our pattern-match system into Haskell. For this purpose, we design a set of typing rules and develop a method for transforming our pattern-match expression to a Haskell program that uses backtracking monad. Chapter 5 presents the method for importing notations of tensor calculus into programming languages. The final chapter summarizes the

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}_i = \begin{pmatrix} a+c \\ b+d \end{pmatrix}_i \qquad \begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}_i = \begin{pmatrix} ac \\ bd \end{pmatrix}_i$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}_j = \begin{pmatrix} a+c & a+d \\ b+c & b+d \end{pmatrix}_{ij} \qquad \begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}_j = \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}_{ij}$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}^i = \begin{pmatrix} a+c & a+d \\ b+c & b+d \end{pmatrix}_i^i \qquad \begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}^i = contract \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}_i^i = ac + bd$$

(a) Symplified index rules for addition (b) Symplified index rules for multiplication

Figure 1.3: Symplified index rules

thesis and discusses future work.

Chapter 2

Design of Pattern-Matching System

2.1 Introduction

Pattern matching is an important feature of programming languages featuring data abstraction mechanisms. Data abstraction serves users with a simple method for handling data structures that contain plenty of complex information. Using pattern matching, programs using data abstraction become concise, human-readable, and maintainable. Most of the recent practical programming languages allow users to extend data abstraction e.g. by defining new types or classes, or by introducing new abstract interfaces. Therefore, a good programming language with pattern matching should allow users to extend its pattern-matching facility akin to the extensibility of data abstraction.

Earlier, pattern-matching systems used to assume a one-to-one correspondence between patterns and data constructors. However, this assumption became problematic when one handles data types whose data have multiple representations. To overcome this problem, Wadler proposed the pattern-matching system views [92] that broke the symmetry between patterns and data constructors. Views enabled users to pattern-match against data represented in many ways. For example, a complex number may be represented either in polar or Cartesian form, and they are convertible to each other. Using views, one can pattern-match a complex number internally represented in polar form with a pattern written in Cartesian form, and vice versa, provided that mutual transformation functions are properly defined. Similarly, one can use the `Cons` pattern to perform pattern matching on lists with joins, where a list `[1,2]` can be either `(Cons 1 (Cons 2 Nil))` or `(Join (Cons 1 Nil) (Cons 2 Nil))`, if one defines a normalization function of lists with join into a sequence of `Cons`.

However, views require data types to have a distinguished canonical form among many possible forms. In the case of lists with join, one can pattern-match with `Cons` because any list with join is canonically reducible to a list with join with the `Cons` constructor at the head. On the other hand, for any list with join, there is no such canonical form that has `Join` at the head. For example, the list `[1,2]` may be decomposed with `Join` into three pairs: `[]` and `[1,2]`, `[1]` and `[2]`, and `[1,2]` and `[]`. For that reason, views do not support pattern matching of lists with join using the `Join` pattern.

Generally, data types without canonical forms are called non-free data types. Mathematically speaking, a non-free data type can be regarded as a quotient on a free data type over an equivalence. An example of non-free data types is, of course, a list with join: it may be viewed as a non-free data type composed of a (free) binary tree equipped with an equivalence between trees with the same leaf nodes enumerated from left to right, such as `(Join Nil (Cons 1 (Cons 2 Nil)))` =

`(Join (Cons 1 Nil) (Cons 2 Nil))`. Other typical examples include sets and multisets, as they are (free) lists with obvious identifications. Generally, as shown for lists with join, pattern matching on non-free data types yields multiple results.* For example, multiset `{1,2,3}` has three decompositions by the `insert` pattern: `insert(1,{2,3})`, `insert(2,{1,3})`, and `insert(3,{1,2})`. Therefore, how to handle multiple pattern-matching results is an extremely important issue when we design a programming language that supports pattern matching for non-free data types.

On the other hand, *pattern guard* is a commonly used technique for filtering such multiple results from pattern matching. Basically, pattern guards are applied after enumerating all pattern-matching results. Therefore, substantial unnecessary enumerations often occur before the application of pattern guards. One simple solution is to break a large pattern into nested patterns to apply pattern guards as early as possible. However, this solution complicates the program and makes it hard to maintain. It is also possible to statically transform the program in a similar manner at the compile time. However, it makes the compiler implementation very complex. *Non-linear pattern* is an alternative method for pattern guard. Non-linear patterns are patterns that allow multiple occurrences of the same variables in a pattern. Compared to pattern guards, they are not only syntactically beautiful but also compiler-friendly. Non-linear patterns are easier to analyze and hence can be implemented efficiently (Section 2.2.1 and 2.3.2). However, it is not obvious how to extend a non-linear pattern-matching system to allow users to define an algorithm to decompose non-free data types. We introduce *extensible pattern matching* to remedy this issue (Section 2.2.2, 2.3.4, and 2.5). Extensibility of pattern matching also enables us to define *predicate patterns*, which are typically implemented as a built-in feature (e.g. pattern guards) in most pattern-matching systems. Additionally, we improve the usability of pattern matching for non-free data types by introducing a syntactic generalization for the `match` expression, called *polymorphic patterns* (Section 2.2.3 and 2.3.3). We also present a non-linear pattern-matching algorithm specialized for backtracking on infinite search trees and supports pattern matching with infinitely many results in addition to keeping efficiency (Section 2.4).

In this chapter, we aim to design a programming language that is oriented toward pattern matching for non-free data types. We summarize the above argument in the form of three criteria that must be fulfilled by a language in order to be used in practice:

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

We believe that the above requirements, called together *criteria of practical pattern matching*, are fundamental for languages with pattern matching. However, none of the existing languages and studies [5, 39, 89, 18] fulfill all of them. In the rest of the chapter, we present a language that satisfies the criteria, together with comparisons with other languages, several working examples, and formal semantics. We emphasize that our proposal has been already implemented in Haskell as the *Egison* programming language, and is open-sourced [30].

*In fact, this phenomenon that "pattern matching against a single value yields multiple results" does not occur for free data types. This is the unique characteristic of non-free data types.

The rest of the chapter is organized as follows. Section 2.2 analyzes the capabilities of the prior work for handling non-free data types and clarifies the problems to be solved. Section 2.3 shows our design of pattern-match expressions that solves the problems introduced in Section 2.2. Section 2.4 explains the internal algorithm for executing pattern-match expressions in Section 2.3. Section 2.5 explains how users customize the pattern-match method for each non-free data type in our system. Section 2.6 shows the formal semantics of our language. Section 2.7 introduces the prior work on pattern-match extensions. Section 2.8 concludes this chapter.

2.2 Motivation

In this section, we discuss the requirements for programming languages to establish practical pattern matching for non-free data types.

2.2.1 Pattern Guards vs. Non-linear Patterns

Compared to pattern guards, non-linear patterns are a compiler-friendly method for filtering multiple matching results efficiently. However, non-linear pattern matching is typically implemented by converting them to pattern guards. For example, some implementations of functional logic programming languages convert non-linear patterns to pattern guards [18, 17, 44]. This method is inefficient because it leads to enumerating unnecessary candidates. In the following program in Curry, `seqN` returns `"Matched"` if the argument list has a sequential `N`-tuple. Otherwise, it returns `"Not matched"`. `insert` is used as a pattern constructor for decomposing data into an element and the rest ignoring the order of elements. We will explain the definition of `insert` in Section 3.5.2.2.

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

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

seq4 (insert x (insert (x+1) (insert (x+2) (insert (x+3) _)))) = "Matched"
seq4 _ = "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
seq4 (take 10 (repeat 0)) -- returns "Not matched" in O(n^4) time
```

When we use a Curry compiler such as PAKCS [4] and KiCS2 [23], we see that "`seq4 (take n (repeat 0))`" takes more time than "`seq3 (take n (repeat 0))`" because `seq3` is compiled to `seq3'` as follows. Therefore, `seq4` enumerates $\binom{n}{4}$ candidates, whereas `seq3` enumerates $\binom{n}{3}$ candidates before filtering the results. If the program uses non-linear patterns as in `seq3`, we easily find that we can check no sequential triples or quadruples exist simply by checking $\binom{n}{2}$ pairs. However, such information is discarded during the program transformation into pattern guards.

```
seq3' (insert x (insert y (insert z _))) | y==x+1 && z==x+2 = "Matched"
seq3' _ = "Not matched"
```

One way to make this program efficient in Curry is to stop using non-linear patterns and instead use a predicate explicitly in pattern guards. The following illustrates such a program.

```
isSeq2 (x:y:rs) = y == x+1
```

```
isSeq3 (x:rs) = isSeq2 (x:rs) && isSeq2 rs

perm [] = []
perm (x:xs) = insert x (perm xs)

seq3 xs | isSeq3 ys = "Matched" where ys = perm xs
seq3 _ = "Not matched"

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

`isSeq2` and `isSeq3` checks whether the elements of the head part of the argument list are in sequence or not. `perm` returns all the permutations of the argument list. In the program, because of laziness, only the head part of the list is evaluated. In addition, because of *sharing* [41], the common head part of the list is pattern-matched only once. Using this call-by-need-like strategy enables efficient pattern matching on sequential n-tuples. However, this strategy sacrifices the readability of programs and makes the program obviously redundant. Instead, we base our work on non-linear patterns and attempt to improve its usability keeping it compiler-friendly and syntactically clean.

2.2.2 Extensible Pattern Matching

As a program gets more complicated, data structures involved in the program get complicated as well. A pattern-matching facility for such data structures (e.g. graphs and mathematical expressions) should be extensible and customizable by users because it is impractical to provide the data structures for these data types as built-in data types in general-purpose languages.

In the studies of computer algebra systems, efficient non-linear pattern-matching algorithms for mathematical expressions that avoid such unnecessary search have already been proposed [2, 58]. Generally, users of such computer algebra systems control the pattern-matching method for mathematical expressions by specifying attributes for each operator. For example, the `Orderless` attribute of the Wolfram language indicates that the order of the arguments of the operator is ignored [3]. However, the set of attributes available is fixed and cannot be changed [1]. This means that the pattern-matching algorithms in such computer algebra systems are specialized only for some specific data types such as multisets. However, there are a number of data types we want to pattern-match other than mathematical expressions, like unordered pairs, trees, and graphs.

Thus, extensible pattern matching for non-free data types is necessary for handling complicated data types such as mathematical expressions. In this chapter, we design a language that allows users to implement efficient backtracking algorithms for general non-free data types by themselves. It provides users with the equivalent power to add new attributes freely by themselves. We discuss this topic again in Section 2.3.4.

2.2.3 Monomorphic Patterns vs Polymorphic Patterns

Polymorphism of patterns is useful for reducing the number of names used as pattern constructors. If patterns are monomorphic, we need to use different names for pattern constructors with similar meanings. As such, monomorphic patterns are error-prone.

For example, the pattern constructor that decomposes a collection into an element and the rest ignoring the order of the elements is bound to the name `insert` in the sample code of Curry [18] as in Section 2.2.1. The same pattern

constructor's name is `Add'` in the sample program of Active Patterns [39]. However, these can be considered as a generalized `cons` pattern constructor for lists to multisets, because they are the same at the point that both of them are a pattern constructor that decomposes a collection into an element and the rest.

Polymorphism is important, especially for value patterns. A value pattern is a pattern that matches when the value in the pattern is equal to the target. It is an important pattern construct for expressing non-linear patterns. If patterns are monomorphic, we need to prepare different notations for value patterns of different data types. For example, we need to have different notations for value patterns for lists and multisets. This is because the equivalence of objects as lists and multisets are not equal although both lists and multisets are represented as a list.

```
pairsAsLists (insert x (insert x _)) = "Matched"
pairsAsLists _ = "Not matched"

pairsAsMultisets (insert x (insert y _)) | (multisetEq x y) = "Matched"
pairsAsMultisets _ = "Not matched"

pairsAsLists [[1,2],[2,1]]     -- returns "Not matched"
pairsAsMultisets [[1,2],[2,1]] -- returns "Matched"
```

2.3 Proposal

In this section, we introduce our pattern-matching system, which satisfies all requirements shown in Section 2.2. Our language has Haskell-like syntax. It is dynamically typed, and as well as Curry, based on lazy evaluation.

2.3.1 The `matchAll` and `match` expressions

We explain the `matchAll` expression. It is a primitive syntax of our language. It supports pattern matching with multiple results.

We show a sample program using `matchAll` in the following. In this thesis, we show the evaluation result of a program in the comment that follows the program. "`--`" is the inline comment delimiter of the proposed language.

```
matchAll [1, 2, 3] as list integer with
  $xs ++ $ys -> (xs, ys)
-- [([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]
```

`matchAll` is composed of an expression called *target*, *matcher*, and *match clause*, which consists of a *pattern* and *body expression*. The `matchAll` expression evaluates the body of the match clause for each pattern-matching result and returns a (lazy) collection that contains all results. In the above code, we pattern-match the target `[1, 2, 3]` as a list of integers using the pattern `$xs ++ $ys`. `list integer` is a matcher to pattern-match the pattern and target as a list of integers. The pattern is constructed using the join pattern constructor. `$xs` and `$ys` are called *pattern variables*. We can use the result of pattern matching referring to them. A `matchAll` expression first consults the matcher on how to pattern-match the given target and the given pattern. Matchers know how to decompose the target following the given pattern and enumerate the results, and `matchAll` then collects the results returned by the matcher. In the sample program, given a join pattern, `list integer` tries to divide a collection into two collections. The collection `[1, 2, 3]` is thus divided into two collections in four ways.

`matchAll` can handle pattern matching that may yield infinitely many results. For example, the following program extracts all twin primes from the infinite list of prime numbers*. We will discuss this mechanism in Section 2.4.2.

```
def twinPrimes :=
  matchAll primes as list integer with
    | _ ++ $p :: #(p + 2) :: _ -> (p, p + 2)

take 6 twinPrimes
-- [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
```

There is another primitive syntax called `match` expression. While `matchAll` returns a collection of all matched results, `match` short-circuits the pattern matching process and immediately returns if any result is found. Another difference from `matchAll` is that it can take multiple match clauses. It tries pattern matching starting from the head of the match clauses, and tries the next clause if it fails. Therefore, `match` is useful when we write conditional branching.

However, `match` is inessential for our language. It is implementable in terms of the `matchAll` expression and macros. The reason is that the `matchAll` expression is evaluated lazily, and, therefore, we can extract the first pattern-matching result from `matchAll` without calculating other pattern-matching results simply by using `car`. We can implement `match` by combining the `matchAll` and `if` expressions using macros. Furthermore, `if` is also implementable in terms of the `matchAll` and `matcher` expression as follows. We will explain the `matcher` expression in Section 2.5. For that reason, we only discuss the `matchAll` expression in the rest of the chapter.

```
defMacro if b e1 e2 :=
  head (matchAll b as matcher
                         $ as something with
                           | True -> [e1]
                           | False -> [e2]
          | $x -> x)
```

2.3.2 Efficient Non-linear Pattern Matching with Backtracking

We have designed our language to handle non-linear patterns efficiently. For example, the calculation time of the following code does not depend on the pattern length. Both of the following examples take $O(n^2)$ time to return the result.

```
matchAll take n (repeat 0) as multiset integer with
  | $x :: #(x + 1) :: _ -> x
-- returns {} in O(n^2) time

matchAll take n (repeat 0) as multiset integer with
  | $x :: #(x + 1) :: #(x + 2) :: _ -> x
-- returns {} in O(n^2) time
```

In our proposal, a pattern is examined from left to right in order, and the binding to a pattern variable can be referred to in the right side of the pattern. In the above examples, the pattern variable `$x` is bound to any element of the collection since the pattern constructor is `insert`. After that, the patterns "`#(x + 1)`" and "`#(x + 2)`" are examined. A pattern that begins with "`#`" is called a *value pattern*. The expression following "`#`" can be any kind of expression. The value patterns match with the target data if the target is equal to the content of the pattern.

*We will explain the meaning of the value pattern `#(p + 2)` and the cons pattern constructor in Section 2.3.2 and 2.3.3, respectively.

Therefore, after successful pattern matching, `$x` is bound to an element that appears multiple times.

We can more elaborately discuss the difference in the efficiency of non-linear patterns and pattern guards in general cases. The time complexity involved in pattern guards is $O(n^{p+v})$ when the pattern matching fails, whereas the time complexity involved in non-linear patterns is $O(n^{p+min(1,v)})$, where n is the size of the target object*, p is the number of pattern variables, and v is the number of value patterns. The difference between v and $min(1, v)$ comes from the mechanism of non-linear pattern matching that backtracks at the first mismatch of the value pattern.

Curry	n=15	n=25	n=30	n=50	n=100
`seq2`	1.18s	1.20s	1.29s	1.53s	2.54s
`seq3`	1.42s	2.10s	2.54s	7.40s	50.66s
`seq4`	3.37s	16.42s	34.19s	229.51s	3667.49s

Egison	n=15	n=25	n=30	n=50	n=100
`seq2`	0.26s	0.34s	0.43s	0.84s	2.72s
`seq3`	0.25s	0.34s	0.46s	0.82s	2.66s
`seq4`	0.25s	0.34s	0.42s	0.78s	2.47s

Table 2.1: Benchmarks of Curry (PAKCS version 2.0.1 and Curry2Prolog(swi 7.6) compiler environment) and Egison (version 3.7.12)

Table 2.1 shows microbenchmark results of non-linear pattern matching for Curry and Egison. The table shows execution times of the Curry program presented in Section 2.2.1 and the corresponding Egison program as shown above. The environment we used was Ubuntu on VirtualBox with 2 processors and 8GB memory hosted on MacBook Pro (2017) with 2.3 GHz Intel Core i5 processor. We can see that the execution times in two implementations follow the theoretical computational complexities discussed above. We emphasize that these benchmark results do not mean Curry is slower than Egison. We can write the efficient programs for the same purpose in Curry if we do not persist in using non-linear patterns. Let us also note that the current implementation of Egison is not tuned up and comparing constant times in two implementations is nonsense.

Value patterns are not only efficient but also easy to read once we are used to them because it enables us to read patterns in the same order the execution process of pattern matching goes. It also reduces the number of new variables introduced in a pattern. We explain the mechanism of how the proposed system executes the above pattern matching efficiently in Section 2.4.

2.3.3 Polymorphic Patterns

The characteristic of the proposed pattern-matching expression is that they take a matcher. This ingredient allows us to use the same pattern constructors for different data types.

For example, one may want to pattern-match a collection `[1, 2, 3]` sometimes as a list and other times as a multiset or a set. For these three types, we can naturally define similar pattern-matching operations. One example is the cons pattern, which is also called `insert` in Section 2.2.1 and 2.3.2. Given a collection,

*Here, we suppose that the number of decompositions by each pattern constructor can be approximated by the size of the target object.

pattern `$x :: $rs` divides it into the "head" element and the rest. When we use the cons pattern for lists, it either yields the result which is uniquely determined by the constructor, or just fails when the list is empty. On the other hand, for multisets, it non-deterministically chooses an element from the given collection and yields many results. By explicitly specifying which matcher is used in match expressions, we can uniformly write such programs in our language:

```
matchAll [1, 2, 3] as list integer with
  | $x :: $rs -> (x, rs)
-- [(1, [2, 3])]
matchAll [1, 2, 3] as multiset integer with
  | $x :: $rs -> (x, rs)
-- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
matchAll [1, 2, 3] as set integer with
  | $x :: $rs -> (x, rs)
-- [(1, [1, 2, 3]), (2, [1, 2, 3]), (3, [1, 2, 3])]
```

In the case of lists, the head element `$x` is simply bound to the first element of the collection. On the other hand, in the case of multisets or sets, the head element can be any element of the collection because we ignore the order of elements. In the case of lists or multisets, the rest elements `$rs` are the collection that is made by removing the "head" element from the original collection. However, 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. This specification is useful, for example, when we pattern-match a graph as a set of edges and enumerate all paths with some fixed length including cycles without redundancy.

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, 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 the polymorphism of patterns, we can use the same syntax for both of them. This greatly improves the readability of the program and makes programming with non-free data types easy.

```
matchAll [1, 2, 3] as list integer with
  | #[2, 1, 3] -> "Matched"
-- []
matchAll [1, 2, 3] as multiset integer with
  | #[2, 1, 3] -> "Matched"
-- {"Matched"}
```

We can pass matchers to a function because matchers are first-class objects. It enables us to utilize polymorphic patterns for defining function. The following is an example utilizing polymorphism of value patterns.

```
def elem/m m x xs :=
  match xs as list m with
    | _ ++ #x :: _ -> True
    | _ -> False
```

2.3.4 Extensible Pattern Matching

In the proposed language, users can describe methods for interpreting patterns in the definition of matchers. Matchers appeared up to here are defined in our language. We show an example of a matcher definition. We will explain the details of this definition in Section 2.5.1.

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

An *unordered pair* is a pair ignoring the order of the elements. For example, the tuple `(2, 5)` is equivalent to `(5, 2)`, if we regard them as unordered pairs. Therefore, the tuple `(2, 5)` is successfully pattern-matched with pattern `(#5, $x)`.

```
matchAll (2, 5) as unorderedPair integer
  | (#5, $x) -> x
-- [2]
```

We can define matchers for more complicated data types. For example, Egi constructed a matcher for mathematical expressions for building a computer algebra system on our language [31, 32]. His computer algebra system is implemented as an application of the proposed pattern-matching system. The matcher for mathematical expressions is used for implementing simplification algorithms of mathematical expressions. A program that converts a mathematical expression object $n\cos^2(\theta) + n\sin^2(\theta)$ to n can be implemented as follows. (Here, we introduced the `mathExpr` matcher and some syntactic sugar for patterns.)

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

2.4 Algorithm

This section explains the pattern-matching algorithm of the proposed system. The formal definition of the algorithm is given in Section 2.6. The method for defining matchers explained in Section 2.5 is deeply related to the algorithm.

2.4.1 Execution Process of Non-linear Pattern Matching

Let us show what happens when the system evaluates the following pattern-matching expression.

```
matchAll [2, 8, 2] as multiset integer with
  | $m :: #m :: _ -> m
-- [2, 2]
```

Figure 2.1 shows one of the execution paths that reaches a matching result. First, the initial *matching state* is generated (step 1). A matching state is a datum that represents an intermediate state of pattern matching. A matching state is a compound type consisting of a stack of *matching atoms*, an environment, and intermediate results of the pattern matching. A matching atom is a tuple of a pattern, a matcher, and an expression called *target*. `MState` denotes the data constructor for matching states. `env` is the environment when the evaluation enters the `matchAll` expression. A stack of matching atoms contains a single matching atom whose pattern, target, and matcher are the arguments of the `matchAll` expression.

In our proposal, pattern matching is implemented as a reduction of matching states. In a reduction step, the top matching atom in the stack of matching

1	`MState [($m :: #m :: _, multiset integer, [2, 8, 2])] env []`

2	`MState [($m, integer, 2), (#m :: _, multiset integer, [8, 2])] env []`
	`MState [($m, integer, 8), (#m :: _, multiset integer, [2, 2])] env []`
	`MState [($m, integer, 2), (#m :: _, multiset integer, [2, 8])] env []`

3	`MState [($m, something, 2), (#m :: _, multiset integer, [8, 2])] env []`
4	`MState [(#m :: _, multiset integer, [8, 2])] env [(m, 2)]`
5	`MState [(#m, integer, 8), (_, multiset integer, [2])] env [(m, 2)]` `MState [(#m, integer, 2), (_, multiset integer, [8])] env [(m, 2)]`
6	`MState [(_, multiset integer, [8])] env [(m, 2)]`
7	`MState [(_, something, [8])] env [(m, 2)]`
8	`MState [] env [(m, 2)]`

Figure 2.1: Reduction path of matching states

atoms is popped out. This matching atom is passed to the procedure called *matching function*. The matching function is a function that takes a matching atom and returns a list of lists of matching atoms. The behavior of the matching function is controlled by the matcher of the argument matching atom. We can control the behavior of the matching function by defining matchers properly. For example, we obtain the following results by passing the matching atom of the initial matching state to the matching function.

```
matchFunction ($m :: #m :: _, multiset integer, [2, 8, 2]) =
  [ [($m, integer, 2), (#m :: _, multiset integer, [8, 2])]
    [($m, integer, 8), (#m :: _, multiset integer, [2, 2])]
    [($m, integer, 2), (#m :: _, multiset integer, [2, 8])] ]
```

Each list of matching atoms is prepended to the stack of the matching atoms. As a result, the number of matching states increases to three (step 2). Our pattern-matching system repeats this step until all the matching states vanish.

For simplicity, in the following, we only examine the reduction of the first matching state in step 2. This matching state is reduced to the matching state shown in step 3. The matcher in the top matching atom in the stack is changed to `something` from `integer`, by definition of `integer` matcher. `something` is the only built-in matcher of our pattern-matching system. `something` can handle only wild-cards or pattern variables, and is used to bind a value to a pattern variable. This matching state is then reduced to the matching state shown in step 4. The top matching atom in the stack is popped out, and a new binding `(m, 2)` is added to the collection of intermediate results. Only `something` can append a new binding to the result of pattern matching.

Similarly to the preceding steps, the matching state is then reduced as shown in step 5, and the number of matching states increases to 2. `#m` is pattern-matched with `8` and `2` by `integer` matcher in the next step. When we pattern-match with a value pattern, the intermediate results of the pattern matching is used as an environment to evaluate it. In this way, "`m`" is evaluated to `2`. Therefore, the first matching state fails to pattern-match and vanishes. The second matching state succeeds in pattern matching and is reduced to the matching state shown

in step 6. In step 7, the matcher is simply converted from `multiset integer` to `something`, by definition of `multiset integer`. Finally, the matching state is reduced to the empty collection (step 8). No new binding is added because the pattern is a wildcard. When the stack of matching atoms is empty, reduction finishes and the matching patching succeeds for this reduction path. The matching result `(m, 2)` is added to the entire result of pattern matching.

We can check the pattern matching for sequential triples and quadruples are also efficiently executed in this algorithm.

2.4.2 Pattern Matching with Infinitely Many Results

The proposed pattern-matching system can eventually enumerate all successful matching results when matching results are infinitely many. It is performed by reducing the matching states in proper order. Suppose the following program:

```
take 8 (matchAll nats as set integer with $m :: $n :: _ -> (m, n))
-- [(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3)]
```

Figure 2.2 shows the search tree of matching states when the system executes the above pattern-matching expression. Rectangles represent matching states, and circles represent final matching states of successful pattern matching. The rectangle at the upper left is the initial matching state. The rectangles in the second row are the matching states generated from the initial matching state one step. Circles o8, r9, and s9 correspond to pattern-matching results `[(m, 1), (n, 1)]`, `[(m, 1), (n, 2)]`, and `[(m, 2), (n, 1)]`, respectively.

One issue on naively searching this search tree is that we cannot enumerate all matching states either in depth-first or breadth-first manners. The reason is that the widths and depths of the search tree can be infinite. Widths can be infinite because a matching state may generate infinitely many matching states (e.g., the width of the second row is infinite), and depths can be infinite when we extend the language with a notion such as recursively defined patterns.

To resolve this issue, we reshape the search tree into a *reduction tree* as presented in Figure 2.3. A node of a reduction tree is a list of matching states, and a node has at most two child nodes, left of which is the matching states generated from the head matching state of the parent, and the right of which is a copy of the tail part of the parent matching states. At each reduction step, the system has a list of nodes. Each row in Figure 2.3 denotes such a list. One reduction step in our system proceeds in the following two steps. First, for each node, it generates a node from the head matching state. Then, it constructs the nodes for the next step by collecting the generated nodes and the copies of the tail parts of the nodes. The index of each node denotes the depth in the tree the node is checked at. Since widths of the tree are at most 2^n for some n at any depth, all nodes can be assigned some finite number, which means all nodes in the tree are eventually checked after a finite number of reduction steps.

We adopt breadth-first search strategy as the default traverse method because there are cases that breadth-first traverse can successfully enumerate all pattern-matching results while depth-first traverse fails to do so when we handle pattern matching with infinitely many results. However, of course, when the size of the reduction tree is finite, the space complexity for depth-first traverse is less expensive. Furthermore, there are cases that the time complexity for depth-first traverse is also less expensive when we extract only the first several successful matches. Therefore, to extend the range of algorithms we can express concisely with pattern matching keeping efficiency, providing users with a method

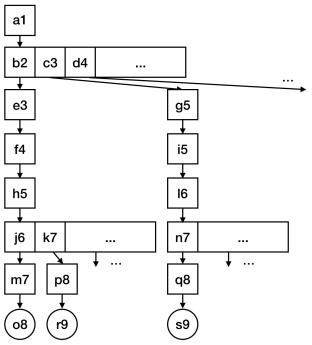

Figure 2.2: Search tree

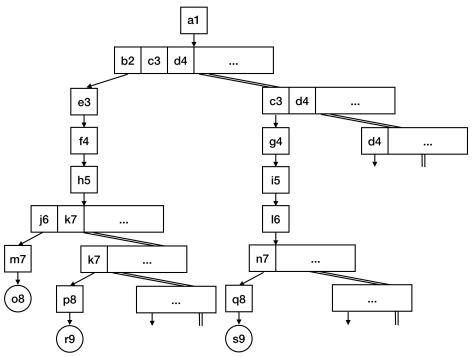

Figure 2.3: Binary reduction tree

for switching search strategy of reduction trees is important. We leave the further investigation of this direction as interesting future work.

2.5 User-Defined Matchers

This section explains how to define matchers from scratch through examples.

2.5.1 Matcher for Unordered Pairs

We explain how the `unorderedPair` matcher shown in Section 2.3.4 works. `unorderedPair` is defined as a function that takes and returns a matcher to specify how to pattern-match against the elements of a pair. 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 the next matchers. A next-target expression is a function that takes a target and returns a list of the next targets. A matcher generates a list of the next matching atoms by combining the next patterns, the next matchers, and a list of next targets. The formal syntax of the `matcher` expression is found in Figure 2.4 in Section 2.6.

`unorderedPair` has two matcher clauses. The primitive-pattern pattern of the first matcher clause is `($, $)`. This matcher clause defines the interpretation of the tuple pattern. This pattern contains two pattern holes `$`. It means that it interprets the first and second elements of the tuple pattern by the matchers specified by the next-matcher expression. In this example, since the next-matcher expression is `(m, m)`, both of the elements of the tuple pattern are pattern-matched using the matcher given by `m`. The primitive-data-match clause of the first matcher clause is `($x, $y) -> [(x, y), (y, x)]`. The pattern `($x, $y)` is pattern-matched with the target datum such as `(2, 5)`, and `$x` and `$y` is matched with `2` and `5`, respectively. The primitive-data-match clause returns `[(2, 5), (5, 2)]`. A primitive-data-match clause returns a collection of *next-targets*. This means the patterns "`#5`" and `$x` are matched with the targets `2` and `5`, or `5` and `2` using the `integer` matcher in the next step, respectively. Pattern matching of primitive-data-patterns is similar to pattern matching against algebraic data types in ordinary functional programming languages. As a result, the first matcher clause

works in the matching function as follows.

```
matchFunction [($x, $y) (unorderedPair integer) (2, 5)] =
  [ [($x, integer, 2), ($y, integer, 5)] [($x, integer, 5), ($y, integer, 2)] ]
```

The second matcher clause is rather simple; this matcher clause simply converts the matcher of the matching atom to the `something` matcher.

2.5.2 Case Study: Matcher for Multisets

As an example of how we can implement matchers for user-defined non-free data types, we show the definition of `multiset` matcher. We can define it simply by using the `list` matcher. `multiset` is defined as a function that takes and returns a matcher.

```
def multiset a :=
  matcher
    | [] as () with
      | [] -> [()]
      | _ -> []
    | $ :: _ as a with
      | $tgt -> tgt
    | $ :: $ as (a, multiset a) with
      | $tgt ->
        matchAll tgt as list a with
          | $hs ++ $x :: $ts -> (x, hs ++ ts)
    | #$val as () with
      | $tgt ->
        match (val, tgt) as (list a, multiset a) with
          | ([], []) -> [()]
          | ($x :: $xs, #x :: #xs) -> [()]
          | (_, _) -> []
    | $ as (something) with
      | $tgt -> [tgt]
```

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

First, we focus on the third 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 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 second matcher clause. The primitive-pattern pattern of the second matcher clause is `$ :: _`. This matcher clause handles a cons pattern whose second argument is a wildcard. We omit the calculation of the next target. In this case, we omit the calculation of the rest elements. The next matcher of this matcher clause is `a` and the next target is `[1,2,3]` when the target is a collection

`[1,2,3]`. We call the optimization technique that omits the calculation of the next target when a pattern contains wildcard *wildcard optimization*.

Next, we focus on the fourth 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. The target collection `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 states 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 states 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 fifth matcher clause. This matcher clause creates the next matching atom by just changing the matcher from `multiset a` to `something`.

2.5.3 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 differences between all the pairs after $(3,11)$ are more than 6.

```
take 10 (matchAll primes as sortedList integer with
         | _ ++ $p :: (_ ++  #(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 $O(n)$.

```
def sortedList a := matcher
  | $ ++ #$px :: $ as (sortedList a, sortedList a) with
    | \$tgt -> matchAll tgt as list a with
               | loop $i (1, $n)
                   ((?(\x -> x < px) & $h_i) :: ...)
```

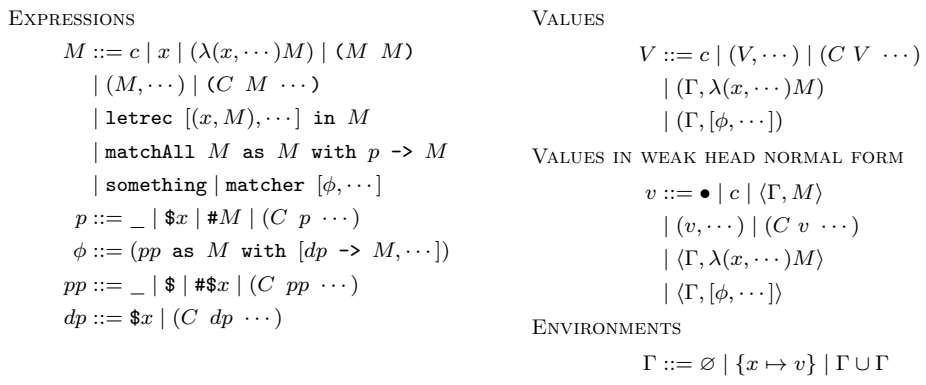

Figure 2.4: Syntax of our language

```
                  (#px :: $ts)
                -> (map (\i -> h_i) [1..n], ts)
  ...
```

We call this optimization technique for nested patterns *pattern fusion*. In the above matcher clause, a loop pattern is used. We will explain loop patterns in Section 3.2.6. Note that pattern fusion 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.

2.6 Formal Semantics

In this section, we present the syntax and big-step semantics of our language (Figure 2.4 and 2.5). We use metavariables $x, y, z, \ldots$, $M, N, \ldots$, $V, W, \ldots$, $v, w, \ldots$, and $p, \ldots$ for variables, expressions, values, values in *weak head normal form*, and patterns respectively. A value is in weak head normal form when only the outermost part has been evaluated, whereas a normal value is fully evaluated. We handle values in weak head normal form for achieving lazy evaluation. In Figure 2.4, c denotes a constant expression and C denotes a data constructor name. $X \cdots$ in Figure 2.4 means a finite list of X. The syntax of our language is similar to that of the Haskell programming language. As explained in Section 2.3.1, $(M, \cdots)$ and $(C\ M\ \cdots)$ denote tuples and data constructions. In this formal language, lists are represented as an algebraic data type using constructors. ϕ, pp, and dp are called matcher clauses, primitive-pattern patterns, and primitive-data patterns respectively. $\Gamma, \Delta, \ldots$ denote variable assignments, i.e., partial functions from variables to values. $\bullet$ denotes the dummy object and is used to implement the recursive definitions by the letrec expressions.

In Figure 2.5, the following notations are used. We write $[a_i]_i$ to mean a lazy list whose elements are evaluated to $[a_1, a_2, \ldots]$. Similarly, $[[a_{ij}]_j]_i$ denotes $[[a_{11}, a_{12}, \ldots], [a_{21}, a_{22}, \ldots], \ldots]$, but each list in the list may have different length. List of tuples $[(a_1, b_1), (a_2, b_2), \ldots]$ may be often written as $[a_i, b_i]_i$ instead of $[(a_i, b_i)]_i$ for short. Concatenation of lists l_1, l_2 are denoted by $l_1 + l_2$, and $a : l$ denotes $[a] + l$ (adding at the front). ϵ denotes the empty list. In general, $\vec{x}$ for some metavariable x is a metavariable denoting a list of what x denotes. However,

Deep Evaluation:

$$\frac{\Gamma, M \downarrow c}{\Gamma, M \Downarrow c}\ \text{D-Constant} \qquad \frac{\Gamma, M \downarrow \langle \Delta, \lambda(x, \cdots) M \rangle}{\Gamma, M \Downarrow (\Delta, \lambda(x, \cdots) M)}\ \text{D-Lambda} \qquad \frac{\Gamma, M \downarrow \langle \Delta, [\phi, \cdots] \rangle}{\Gamma, M \Downarrow (\Delta, [\phi, \cdots])}\ \text{D-Matcher}$$

$$\frac{\Gamma, M \downarrow (\langle \Delta, N_i \rangle)_i \qquad \Delta, N_i \Downarrow V_i (\forall i)}{\Gamma, M \Downarrow (V_i)_i}\ \text{D-Tuple} \qquad \frac{\Gamma, M \downarrow (C\ \langle \Delta, N_i \rangle)_i \qquad \Delta, N_i \Downarrow V_i (\forall i)}{\Gamma, M \Downarrow (C\ V_i)_i}\ \text{D-Cons}$$

Core features as a functional language:

$$\frac{}{\Gamma, c \downarrow c}\ \textsc{Constant} \qquad \frac{}{\Gamma \cup \{x \mapsto v\} \cup \Delta, x \downarrow v}\ \textsc{Variable} \qquad \frac{}{\Gamma, (M_i)_i \downarrow (\langle \Gamma, M_i \rangle)_i}\ \textsc{Tuple}$$

$$\frac{}{\Gamma, (C\ M_i)_i \downarrow (C\ \langle \Gamma, M_i \rangle)_i}\ \textsc{Cons} \qquad \frac{}{\Gamma, (\lambda(x_i)_i M) \downarrow (\Gamma, \lambda(x_i)_i M)}\ \textsc{Lambda}$$

$$\frac{\Gamma, M \downarrow (\Delta, \lambda(x_i)_i M') \qquad \Gamma, N \downarrow (v_i)_i \qquad \Delta \cup \{x_i \mapsto v_i\}_i, M' \downarrow w}{\Gamma, (M\ N) \downarrow w}\ \textsc{Application}$$

$$\frac{\Gamma \cup \{x_i \mapsto \bullet_i\}_i, M_j \downarrow \langle \Delta, M_j \rangle \quad (\forall j) \qquad \Delta' = \mathit{update}(\Delta, [\bullet_i]_i, [v_i]_i) \qquad \Delta', N \downarrow w}{\Gamma, \texttt{letrec}\ [(x_i, M_i)]_i\ \texttt{in}\ N \downarrow w}\ \textsc{Letrec}$$

Evaluation of `matcher` and `matchAll`:

$$\frac{}{\Gamma, \texttt{matcher}\ [pp_i\ \texttt{as}\ M_i\ \texttt{with}\ [dp_{ij}\ \texttt{->}\ N_{ij}]_j]_i \downarrow \langle \Gamma, [pp_i, M_i, [dp_{ij}, N_{ij}]_j]_i \rangle}\ \textsc{Matcher}$$

$$\frac{\Gamma, M \downarrow v \qquad \Gamma, N \downarrow m \qquad [[[p \sim_m v], \Gamma, \varnothing]] \Rrightarrow [\Delta_i]_i \qquad \Gamma \cup \Delta_i, L \downarrow v_i \quad (\forall i)}{\Gamma, \texttt{matchAll}\ M\ \texttt{as}\ N\ \texttt{with}\ p\ \texttt{->}\ L \downarrow [v_i]_i}\ \textsc{MatchAll}$$

Matching states:

$$\frac{}{\epsilon \Rrightarrow \epsilon}\ \text{MS-Bfs-Nil} \qquad \frac{\vec{s} \Rrightarrow \vec{\Gamma}, \vec{s'} \qquad \vec{s'} \Rrightarrow \vec{\Delta}}{\vec{s} \Rrightarrow \vec{\Gamma} + \vec{\Delta}}\ \text{MS-Bfs-Cons}$$

$$\frac{\vec{s}_i \to \texttt{opt}\ \Gamma_i, \texttt{opt}\ \vec{s'}_i, \texttt{opt}\ \vec{s''}_i \quad (\forall i)}{[\vec{s}_i]_i \Rrightarrow \textstyle\sum_i (\texttt{opt}\ \Gamma_i), \sum_i (\texttt{opt}\ \vec{s'}_i) + \sum_i (\texttt{opt}\ \vec{s''}_i)}\ \text{MS-Step}$$

$$\frac{}{\epsilon \to \texttt{none}, \texttt{none}, \texttt{none}}\ \text{MS-Fail} \qquad \frac{}{(\epsilon, \Gamma, \Delta) : \vec{s} \to (\texttt{some}\ \Delta), \texttt{none}, (\texttt{some}\ \vec{s})}\ \text{MS-Success}$$

$$\frac{p \sim_m^{\Gamma \cup \Delta} v \downarrow [\vec{a}_i]_i, \Delta'}{((p \sim_m v) : \vec{a}, \Gamma, \Delta) : \vec{s} \to \texttt{none}, (\texttt{some}[\vec{a}_i + \vec{a}, \Gamma, \Delta \cup \Delta']_i), (\texttt{some}\ \vec{s})}\ \text{MS-Top-MAtom}$$

Matching atoms:

$$\frac{}{\_ \sim^{\Gamma}_{\texttt{something}} v \downarrow [\epsilon], \varnothing}\ \text{MA-Something-WC} \qquad \frac{}{\texttt{\$x} \sim^{\Gamma}_{\texttt{something}} v \downarrow [\epsilon], \{x \mapsto v\}}\ \text{MA-Something-PatVar}$$

$$\frac{pp \approx^{\Gamma} p \downarrow \textbf{fail} \qquad p \sim^{\Gamma}_{(\vec{\phi}, \Delta)} v \downarrow \vec{\vec{a}}, \Gamma'}{p \sim^{\Gamma}_{((pp, M, \vec{\sigma}) : \vec{\phi}, \Delta)} v \downarrow \vec{\vec{a}}, \Gamma'}\ \text{MA-PP-Fail}$$

$$\frac{pp \approx^{\Gamma} p \downarrow [p'_i]_i, \Delta' \qquad dp \approx v \downarrow \textbf{fail} \qquad p \sim^{\Gamma}_{((pp, M, \vec{\sigma}) : \vec{\phi}, \Delta)} v \downarrow \vec{\vec{a}}, \Gamma'}{p \sim^{\Gamma}_{((pp, M, (dp, N) : \vec{\sigma}) : \vec{\phi}, \Delta)} v \downarrow \vec{\vec{a}}, \Gamma'}\ \text{MA-DP-Fail}$$

$$\frac{pp \approx^{\Gamma} p \downarrow [p'_j]_j, \Delta' \qquad dp \approx v \downarrow \Delta'' \qquad \Delta \cup \Delta' \cup \Delta'', N \downarrow [[v'_{ij}]_j]_i \qquad \Delta, M \downarrow [m'_j]_j}{p \sim^{\Gamma}_{((pp, M, (dp, N) : \vec{\sigma}) : \vec{\phi}, \Delta)} v \downarrow [[p'_j \sim_{m'_j} v'_{ij}]_j]_i, \varnothing}\ \text{MA-Step}$$

Pattern matching on patterns:

$$\frac{}{\_ \approx^{\Gamma} p \downarrow [], \varnothing}\ \text{PPP-WC} \qquad \frac{}{\texttt{\$} \approx^{\Gamma} p \downarrow [p], \varnothing}\ \text{PPP-PatHole} \qquad \frac{\Gamma, M \downarrow v}{\texttt{\#\$y} \approx^{\Gamma} \texttt{\#}M \downarrow \epsilon, \{y \mapsto v\}}\ \text{PPP-ValPat}$$

$$\frac{pp_i \approx^{\Gamma} p_i \downarrow \vec{p}_i, \Gamma_i \quad (\forall i)}{(\texttt{C}\ pp_1 \ldots pp_n) \approx^{\Gamma} (\texttt{C}\ p_1 \ldots p_n) \downarrow \textstyle\sum_i \vec{p}_i, \bigcup_i \Gamma_i}\ \textsc{PPP-Constructor}$$

Pattern matching on data:

$$\frac{}{\texttt{\$z} \approx v \downarrow \{z \mapsto v\}}\ \text{PDP-PatVar} \qquad \frac{dp_i \approx v_i \downarrow \Gamma_i \quad (\forall i)}{(\texttt{C}\ dp_1 \ldots dp_n) \approx (\texttt{C}\ v_1 \ldots v_n) \downarrow \bigcup_i \Gamma_i}\ \textsc{PDP-Constructor}$$

Figure 2.5: Formal semantics of our language

we do *not* mean by $\vec{x}_i$ the i-th element of $\vec{x}$; if we write $[\vec{x}_i]_i$, we mean a list of a list of x.

Our language has some special primitive types: matching atoms $a, \ldots$, matching states $s, \ldots$, primitive-data-match clauses $\sigma, \ldots$, and matchers $m, \ldots$. A matching atom consists of a pattern p, a matcher m, and a value v, and written as $p \sim_m v$. A matching state is a tuple of a list of matching atoms and two variable assignments. A primitive-data-match clause is a tuple of a primitive-data pattern and an expression, and a matcher clause is a tuple of a primitive-pattern pattern, an expression, and a list of data-pattern clauses. A matcher is a pair containing a list of matcher clauses and a variable assignment. Note that matchers, matching states, etc. are all values.

The first part of Figure 2.5 defines the semantics for evaluating values in weak head normal form to the full-evaluated values. The judgment $\Gamma, M \downarrow v$ denotes that the given expression M is evaluated to v, which is a value in weak head normal form, under the environment Γ. The judgment $\Gamma, M \Downarrow V$ denotes that M is evaluated to the full-evaluated value v under Γ. In the evaluation rules D-Tuple and D-Cons, we can see that the subexpressions inside tuples and constructor data are deeply evaluated using the judgment $\Delta, N_i \Downarrow V_i$ recursively.

The second part of Figure 2.5 defines the semantics of the basic features of functional languages. The semantics for all the expressions in this part are defined in the common method. The first judgment in the assumption of Letrec evaluates each expression in bindings after assigning the variables in letrec to dummy values. Then, the second judgment in the assumption of Letrec updates the dummy values with these evaluated values. The function $update(\Delta, [v_i]_i, [w_i]_i)$ returns the new environment Δ' by updating the values that are stored at the positions where each v_i is located with w_i.

In the figure, MatchAll and Matcher show the definition of evaluation of `matcher` and `matchAll` expressions, respectively. Evaluation results of expressions are specified by the judgment $\Gamma, e \downarrow \vec{v}$, which denotes given a variable assignment Γ and an expression e one gets a list of values $\vec{v}$. The definition of `matchAll` relies on another type of the judgment $\vec{\vec{s}} \Rrightarrow \vec{\Gamma}$, which defines how the search space is examined.

The rules that start with MS- in the second part of Figure 2.5 show the definition of the judgment $\vec{\vec{s}} \Rrightarrow \vec{\Gamma}$. The judgment $\vec{\vec{s}} \Rrightarrow \vec{\Gamma}$ takes a list of lists of matching states and returns pattern-match results. We handle a list of lists of matching states for traversing a search tree in breadth-first order as explained in Section 2.4.2. This list of lists of matching states represents the list of nodes at the same depth level in the binary reduction tree in Figure 2.3. For example, $\vec{\vec{s}}$ is $[[a1]]$ at the first depth level, $[[b2, c3, d4, \cdots]]$ at the second level, and $[[e3], [c3, d4, \cdots]]$ at the third depth level in the binary reduction tree in Figure 2.3. MS-Bfs-Nil and MS-Bfs-Cons define this breadth-first search. In MS-Bfs-Cons, $\Rrightarrow$ is inductively defined using $\vec{\vec{s}} \Rightarrow \vec{\Gamma}, \vec{\vec{s'}}$, which is defined by MS-Step. The judgment $\vec{\vec{s}} \Rightarrow \vec{\Gamma}, \vec{\vec{s'}}$ takes a list of lists of matching states in the same depth level and returns the list of nodes in the next depth level. $\vec{\Gamma}$ represents the pattern-match results, and $\vec{\vec{s'}}$ represents the list of nodes in the next depth level. For example, in the binary reduction tree in Figure 2.3, when $\vec{\vec{s}}$ is $[[e3], [c3, d4, \cdots]]$, Γ and $\vec{\vec{s'}}$ are $\varnothing$ and $[[f4], [g4], [d4, \cdots]]$, respectively. MS-Fail, MS-Success, and MS-Top-MAtom define the evaluation of the judgment $\vec{s} \rightarrow \texttt{opt}\,\Gamma, \texttt{opt}\,\vec{s'}, \texttt{opt}\,\vec{s''}$. In these rules, we introduce notations for (meta-level) option types. `none` and `some` x are the constructors of the option type, and `opt` x is a metavariable for an

optional value (possibly) containing what the metavariable x denotes. $\sum_i(\texttt{opt}\, x_i)$ creates a list by collecting all the valid (non-`none`) x_i preserving the order. In the judgment $\vec{s} \to \texttt{opt}\,\Gamma, \texttt{opt}\,\vec{s'}, \texttt{opt}\,\vec{s''}$, $\texttt{opt}\,\Gamma$ represents a pattern-match result, $\texttt{opt}\,\vec{s'}$ represents next matching states, and $\texttt{opt}\,\vec{s''}$ represents the rest of matching states. This judgment takes a list of matching states (a node of the binary reduction tree) and returns the next matching states. MS-Fail defines that pattern matching fails when the next matching states of the previous matching state is empty. MS-Success defines that pattern matching succeeds when the stack of matching atoms becomes empty. MS-Top-MAtom expands the top matching atom of the first matching state by the method defined in the matcher of the top matching atom.

The rules that start with MA- in the third part of Figure 2.5 show the definition of evaluation of a matching atom. The judgment $p \sim_m^\Gamma v \downarrow \vec{\vec{a}}, \Delta$ is a 6-ary relation. One reads this judgment "performing pattern matching on v against p using the matcher m under the variable assignment Γ yields the next matching atoms $\vec{\vec{a}}$ and the variable assignment Δ." $\vec{\vec{a}}$ being empty means the pattern matching failed. If $[\epsilon]$ is returned as $\vec{\vec{a}}$, it means the pattern matching succeeded and no further search is necessary. These rules define the matching function explained in Section 2.4.1. MA-Something-WC and MA-Something-PatVar define the pattern-match method for the only built-in matcher `something`. The `something` matcher can handle only a wildcard or a pattern variable and always succeeds in pattern matching. When the pattern is a pattern variable, a new assignment is added. MA-PP-Fail, MA-DP-Fail, MA-Step handle a user-defined matcher. As explained in Section 2.5, one needs to pattern-match patterns and data to define user-defined matchers. Their formal definitions are given by judgments $pp \approx^\Gamma p \downarrow \vec{p'}, \Delta$ and $dp \approx v \downarrow \Gamma$ that are defined in the fourth part and the last part of Figure 2.5. MA-PP-Fail defines that we try the next matcher clauses when pattern matching for primitive-pattern pattern fails. MA-DP-Fail defines that we try the next primitive-data-match clauses when pattern matching for primitive-data pattern fails. MA-Step defines that we get the next matching atoms by executing the body of the primitive-data match clause where pattern matching for the primitive-pattern pattern and the primitive-data pattern succeeds.

The rules that start with PPP- in the fourth part of Figure 2.5 show the definition of pattern matching for primitive-pattern patterns. One reads the judgment $pp \approx^\Gamma p \downarrow \vec{p'}, \Delta$ "performing pattern matching on the pattern p against the primitive-pattern pattern pp under the environment Γ yields the next patterns $\vec{p'}$ and the variable assignment Δ". PPP-WC, PPP-PatHole, PPP-ValPat, and PPP-Constructor define primitive-pattern-match for a wildcard, a pattern hole, a primitive value pattern, a constructor pattern, respectively.

The rules that start with PDP- in the last part of Figure 2.5 show the definition of pattern matching for primitive-data patterns. One reads the judgment $dp \approx v \downarrow \Gamma$ "performing pattern matching on v against the primitive-data pattern dp yields the variable assignment Γ". PDP-PatVar and PDP-Constructor define primitive-data-match for a pattern variable and a constructor pattern, respectively.

2.7 Related Work

When pattern matching first appeared, pattern matching could only be applied to the specific types of algebraic data types [24]. Huge efforts have been conducted to remove this limitation [48, 25]. 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.

2.7.1 1980s: Spread of Pattern Matching and Invention of Views

From this decade, functional languages with user-defined algebraic data types and pattern matching for them became common. Miranda by Turner [90] and Haskell [48] were the most popular among these languages, and the first pattern-match extensions for widening the target of pattern matching beyond algebraic data types were designed on them.

Miranda's laws [84, 85] and Wadler's views [92, 65] 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 [92] presents 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 support neither non-linear patterns nor pattern matching with multiple results. Views are supported as a GHC extension in Haskell [5]. Views are implemented also in Racket [86].

At the same time, more expressive pattern matching is explored by Queinnec [70], who proposed 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-match-oriented definition of the basic list processing functions. For example, the following `member` definition is presented in Queinnec's paper [70].

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

In Queinnec's language, we represent a wildcard by `-`. Pattern variables are prepended by `?` or `??`. A pattern variable that starts with `??` appears only in a list pattern. This pattern variable matches a part of the target list.

2.7.2 1990s and 2000s: Exploration for Expressive Patterns

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

Erwig's active patterns [39, 80] 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. An example of pattern matching against graphs using matching function is also shown in [40]. `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 hiding the recursion as follows.

```
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.

Tullsen's first-class patterns [89] are another extension of views. First-class patterns support pattern matching with multiple results. In first-class patterns, we can define pattern constructors that have multiple decompositions. 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.

2.7.3 2010s: Toward a Unified Theory of Pattern-Match-Oriented Programming

In this decade, a unified theory for practical pattern matching for non-free data types has been pursued. Egison proposed in this thesis is such research. The research listed and organized the properties for practical pattern matching for non-free data types. We proposed three criteria. The criteria are as follows: (1) Efficiency of the backtracking algorithm for non-linear patterns; (2) Ad-hoc polymorphism of patterns; (3) Extensibility of pattern matching.

2.8 Conclusion

We designed a user-customizable efficient non-linear pattern-matching system by regarding pattern matching as reduction of matching states that have a stack of matching atoms and intermediate results of pattern matching. This system enables us to concisely describe a wide range of programs, especially when non-free data types are involved. For example, our pattern matching architecture is useful to implement a computer algebra system because it enables us to directly pattern-match mathematical expressions and rewrite them.

The major significance of our pattern matching system is that it greatly improves the expressivity of the programming language by allowing programmers to freely extend the process of pattern matching by themselves. Although we consider that the current syntax of matcher definition is already clean enough, we leave further refinement of the syntax of our surface language as future work.

We believe the direct and concise representation of algorithms enables us to implement really new things that go beyond what was considered practical before. We hope our work will lead to breakthroughs in various fields. ,

Chapter 3

Pattern-Matching-Oriented Programming Style

3.1 Overview

Throughout the history of functional programming, recursion has emerged 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.

In this chapter, we advocate a new programming paradigm called *pattern-match-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, we introduce many programming techniques that replace explicit recursions with an intuitive pattern by confining recursions inside patterns. We classify these techniques as pattern-match-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.

This chapter is organized as follows. Section 3.2 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. Section 3.3 catalogs pattern-match-oriented programming techniques utilizing the features introduced in Section 3.2. Section 3.4 explores the effect of pattern-match-oriented programming in more practical situations. Section 3.5 reviews the related work. Finally, Section 3.6 concludes this chapter.

3.2 Quick Tour of the Egison Pattern-Match-Oriented Language

This section quickly introduces the pattern-matching facility of Egison.

3.2.1 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 a pair of identical elements.

```
matchAll [1, 2, 3, 2, 4, 3] as list integer with
| _ ++ $x :: _ ++ #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.

The variables inside a value pattern refer to a value bound outside the pattern-match expression when the corresponding pattern variables do not appear on the left-side of a pattern. For example, the value pattern `#x` refers to the value bound by the first argument of `delete` in the following sample.

```
def delete x xs :=
  match xs as list eq with
    | $hs ++ #x :: $ts -> hs ++ ts
    | _ -> xs

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

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 in order.

```
def twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: #(p + 2) :: _ -> (p, p + 2)

take 8 twinPrimes
-- [(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 `?`.

```
def twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: ?(\q -> q = p + 2) :: _ -> (p, p + 2)
```

3.2.2 Non-linear Pattern Matching with Backtracking

The pattern-matching algorithm inside Egison includes the backtracking mechanism for efficient non-linear pattern matching.

```
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ -> x
-- returns [] in O(n^2) time
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ ++ #x :: _ -> x
-- returns [] in O(n^2) time
```

The above expressions match a collection that consists of integers from 1 to n as a list of integers for enumerating identical pairs and triples, respectively. This target collection contains neither identical pairs nor triples. Therefore both expressions return an empty collection.

When evaluating the second expression, Egison interpreter does not try pattern matching for the second `#x` because pattern matching for the first `#x` always fails. Therefore, the time complexities of the above expressions are identical. The pattern-matching algorithm inside Egison is discussed in Section 2.4 in detail.

3.2.3 Ad-hoc Polymorphism of Patterns by Matchers

Another merit of matchers, in addition to the extensibility of pattern-matching algorithms, is the ad-hoc polymorphism of patterns. The ad-hoc polymorphism of patterns is 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 multisets, the cons pattern decomposes a collection into an element and the rest elements ignoring the order of the elements. In the case of sets, the rest elements are the same as the original collection because we ignore the redundant elements. If we regard a set as a collection that contains infinitely many copies of each element, this specification of the cons pattern for sets is natural.

```
matchAll [1,2,3] as list something with $x :: $xs -> (x,xs)
-- [(1,[2,3])]
matchAll [1,2,3] as multiset something with $x :: $xs -> (x,xs)
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]
matchAll [1,2,3] as set something with $x :: $xs -> (x,xs)
-- [(1,[1,2,3]),(2,[1,2,3]),(3,[1,2,3])]
```

Polymorphic patterns are useful especially for 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 lists but true if we regard them as multisets. Still, thanks to ad-hoc 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.

```
matchAll [1,2,3] as list integer with #[2,1,3] -> "Matched" -- []
matchAll [1,2,3] as multiset integer with #[2,1,3] -> "Matched" -- ["Matched"]
```

3.2.4 `matchAllDFS` for Controlling the Order of Pattern-Matching Process

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 (Section 5.2 of [38]). As a result, the order of the pattern-matching results is as follows.

```
take 8 (matchAll [1..] as set something with
        | $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(2,3),(3,2)]
```

The above order is preferable for traversing an infinitely large reduction tree. However, sometimes, this order is not preferable (see Section 3.3.1.2 and Section 3.3.4.1). `matchAllDFS` that traverses a reduction tree in depth-first order is provided for this reason.

```
take 8 (matchAllDFS [1..] as set something with
        | $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]
```

3.2.5 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 triples* as an example of and-patterns and or-patterns. A prime triple is a triple 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 $p+4$.

```
def primeTriples := matchAll primes as list integer with
  | _ ++ $p :: ((#(p + 2) | #(p + 4)) & $m) :: #(p + 6) :: _
  -> (p, m, p + 6)

take 6 primeTriples -- [(5,7,11),(7,11,13),(11,13,17),(13,17,19),(17,19,23)
    ,(37,41,43)]
```

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 (matchAll primes as list integer with
         | _ ++ $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)]
```

3.2.6 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 [33].

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.

```
def comb2 xs := matchAll xs as list something with
  | _ ++ $x_1 :: _ ++ $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.

```
def comb n xs := matchAll xs as list something with
             | loop $i                -- index variable
                    (1, n)            -- index range
                    (_ ++ $x_i :: ...) -- repeat pattern
                    _                 -- final pattern
             -> map (\i -> x_i) [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 *final 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. An index range is a tuple of an *initial number* and *final number*. A repeat pattern is a pattern repeated when the index variable is in the index range. A final pattern is a pattern expanded when the index variable gets out of the index range. Inside loop patterns, we can use the *ellipsis pattern* $(\ldots)$. The repeat pattern or the final 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. For example, when $n = 3$, the above loop pattern is expanded as follows.

```
loop $i (1, 3) (_ ++ $x_i :: ...) _ -->
_ ++ $x_1 :: (loop $i (2, 3) (_ ++ $x_i :: ...) _) -->
_ ++ $x_1 :: _ ++ $x_2 :: (loop $i (3, 3) (_ ++ $x_i :: ...) _) -->
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: (loop $i (4, 3) (_ ++ $x_i :: ...) _) -->
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: _
```

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 final number. When the final number is a pattern, the ellipsis pattern is replaced with both the repeat pattern and the final pattern, and the repeat count when the ellipsis pattern is replaced with the final pattern is pattern-matched with that pattern. The following loop pattern enumerates all initial prefixes of the target collection.

```
matchAll [1,2,3,4] as list something with
| loop $i (1, $n) ($x_i :: ...) _ -> map (\i -> x_i) [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 Section 3.3.4.1. More formal specification of syntax and semantics of loop patterns is shown in the author's previous paper [33].

3.2.7 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 the second element in order.

```
matchAll [2,3,1,4,5] as list integer with
| {     @    ::     @    :: $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 Section 3.3.3.

Some readers might wonder that a sequential pattern can be transformed into a nested `matchAll` expression. There are at least two reasons why it is impossible. First, a nested `matchAll` expression breaks breadth-first search strategy: the inner `matchAll` for the second result of the outer `matchAll` is executed only after the inner `matchAll` for the first result of the outer `matchAll` 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 `matchAll` 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 `matchAll` expressions syntactically.

3.2.8 Matcher Compositions

Matchers are composable. 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 Section 3.3.3.

```
def intersect xs ys := matchAll (xs,ys) as (set eq, set eq) with
  | ($x :: _, #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.*

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.

```
def 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.

```
def adjacencyGraph := multiset (integer, multiset integer)
```

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

*A definition of the `eq` matcher is explained in Section 6.3 of [38].

```
def binaryTree a := algebraicDataMatcher
  | bLeaf a
  | bNode a (binaryTree a) (binaryTree 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 Section 3.3.4.1.

```
def tree a := algebraicDataMatcher
  | leaf a
  | node a (multiset (tree a))
```

3.3 Pattern-Match-Oriented Programming Design Patterns

This section introduces basic pattern-match-oriented programming techniques that replace explicit recursions with 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 showing situations in which they are useful. The following table shows this list.

Name	Description	Explained and Used in
Join-cons pattern for list	Enumerate combinations of elements.	3.3.1
Cons pattern for multiset	Enumerate permutations of elements.	3.3.2, 3.3.3, 3.3.4, 3.4.1, 3.4.3, 3.4.4
Tuple pattern for collections	Compare multiple collections.	3.3.3, 3.3.4, 3.4.1, 3.4.4
Loop pattern	Describe repetitions inside patterns.	3.3.4

3.3.1 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.

3.3.1.1 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 our natural explanation of `map`.

```
def map f xs := matchAll xs as list something with
  | _ ++ $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.

```
def filter pred xs := matchAll xs as list something with
  | _ ++ (and ?pred $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 `head (matchAll ...)` because Egison evaluates `matchAll` lazily.*

```
def elem x xs := match xs as list eq with
  | _ ++ #x :: _ -> True
  | _            -> False
```

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

```
def delete x xs := match xs as list eq with
  | $hs ++ #x :: $ts -> hs ++ ts
  | _                -> xs
```

The predicate `any` and `every` [76] 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.

```
def any pred xs := match xs as list something with
  | _ ++ ?pred :: _ -> True
  | _               -> False

def every pred xs := match xs as list something with
  | _ ++ !?pred :: _ -> False
  | _                -> True
```

3.3.1.2 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-match-oriented style as follows.

```
def unique xs := matchAllDFS xs as list eq with
  | _ ++ $x :: !(_ ++ #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.

```
def unique xs := matchAllDFS xs as list eq with
  | $hs ++ (!?(\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.

*`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 `++` ....

```
def unique xs := matchAllDFS xs as list eq with
  |  {@               ++ $x :: _,
      !(_ ++ #x :: _)}
  -> x
```

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

```
def concat xss := matchAllDFS xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x
```

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

```
matchAll [[1..], (map negate [1..])] as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x
--  [1, 2, -1, 3, -2, 4, -3, 5, -4, 6]
```

3.3.2 Cons Patterns for Multisets

Cons patterns for a multiset are useful when we want to treat a collection ignoring the order of elements. We often meet such a situation, especially when describing mathematical algorithms.

We start from a simple example. The `lookup` function for association lists can be defined using a single cons pattern for multiset. A single cons pattern for a multiset can be replaced by a join-cons pattern for a list.

```
def lookup k ls := match ls as multiset (eq, something) with
  | (#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 $P(n,k) = \frac{n!}{(n-k)!}$ permutations of k elements, whereas join-cons patterns can be used to enumerate $C(n,k) = \frac{n!}{k!(n-k)!}$ combinations of k elements.

```
matchAll [1,2,3] as list integer with
  | _ ++ $x :: _ ++ $y :: _ -> (x,y)
-- [[1,2],[1,3],[2,3]]

matchAll [1,2,3] as multiset integer with
  | $x :: $y :: _ -> (x,y)
-- [(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
```

The descriptions of algorithms for which nested cons patterns for multisets 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-match-oriented programming demonstrates its full power. The rest of this chapter

discusses how we can describe this wide variety of patterns for multisets by just combining pattern constructs introduced in Section 3.2.

3.3.3 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 Section 3.2.8.

```
def difference xs ys := matchAll (xs, ys) as (set eq, set eq) with
  | ($x :: _, !(#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 the 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 remainder of the two collections. Sequential not-patterns often appear in mathematical algorithms, and we show an example again in Section 3.4.1.

```
def singleCommonElem := match (xs, ys) as (multiset eq, multiset eq) with
                     | [($x :: @, #x :: @),
                        !($y :: _, #y :: _)] -> True
                     | _ -> False
```

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

```
match (xs, ys) as (list eq, list eq) with
  | loop $i (1,$n)
      [($x_i :: @, #x_i :: @), [...]]
      !($y :: _, #y :: _)
  -> map (\i -> x_i) [1..n]
```

3.3.4 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 (Section 3.3.4.1) and when the number of pattern variables that appear in a pattern changes by parameters (Section 3.3.4.2). 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.

3.3.4.1 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 Section 3.2.8. 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-match-oriented"` category, and the `"Dynamically typed"` sub-category of the `"Functional programming"` category.

```
def treeData :=
  Node "Programming language"
    [Node "pattern-match-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"]],
     Node "Logic programming" [Leaf "Prolog", 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. The ellipsis pattern in this loop pattern is not placed in 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.

```
def ancestors x t := matchAll t as tree string with
  | loop $i (1,$n)
      (node $c_i (... ::  _))
      (leaf #x)
  -> map (\i -> c_i) [1..n]

ancestors "Egison" treeData
-- [["Programming language", "pattern-match-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.

```
def descendants x t := matchAllDFS t as tree string with
  | loop _ (1,_)
      (node _ (... ::  _))
      (node #x ((loop _ (1,_)
                    (node _ (... :: _))
                    (leaf $y)) :: _))
  -> y

descendants "Functional language" treeData
-- ["OCaml", "Haskell", "Curry", "Coq", "Egison", "Lisp", "Scheme", "Racket"]
```

Egison is more elegant than XML path language [7] 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 enumerate all ancestors of a node, for example.

3.3.4.2 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 $a_3 \pm 1 \neq a_4$.

```
matchAll [1,2,3,4] as multiset integer with
  $a_1 ::
   (!#(a_1 - 1) & !#(a_1 + 1) & $a_2) ::
    (!#(a_1 - 2) & !#(a_1 + 2) & !#(a_2 - 1) & !#(a_2 + 1) & $a_3) ::
     (!#(a_1 - 3) & !#(a_1 + 3) & !#(a_2 - 2) & !#(a_2 + 2) & !#(a_3 - 1) & !#(
    a_3 + 1) & $a_4) ::
      [] -> [a_1,a_2,a_3,a_4]
-- [[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 to 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 is effectively used for representing this pattern.

```
def nQueens n :=
  matchAll [1..n] as multiset integer with
  | $a_1 ::
     (loop $i (2,n)
           ((loop $j (1, i - 1)
                  (!#(a_j - (i - j)) & !#(a_j + (i - j)) & ...)
                  $a_i) :: ...)
           [] -> map (\i -> a_i) [1..n]

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

3.4 Pattern-Match-Oriented Programming in More Practical Situations

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

3.4.1 SAT Solver

To see the effect of pattern-match-oriented programming for implementing practical algorithms, we implement a SAT solver. A SAT solver determines whether a given propositional logic formula has an assignment for which the formula evaluates 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, $(p \vee q) \wedge (\neg p \vee r) \wedge (\neg p \vee \neg r)$ is a formula in conjunctive normal form that has a solution; $p =$ `False`, $q =$ `True`, and $r =$ `True`.

3.4.1.1 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. Furthermore, we represent a literal as an integer. We represent positive and negative literals as positive and negative integers respectively: for example, p and $\neg p$ are represented as 1 and -1, respectively. 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[46]. The `dp` function takes a list of propositional variables and a logical formula, and returns `True` if there is a solution, otherwise returns `False`.

```
def dp vars cnf :=
  match (vars, cnf) as (multiset integer, multiset (multiset integer)) with
  | (_, []) -> True
  | (_, [] :: _) -> False
  -- 1-literal rule
  | (_, ($l :: []) :: _) -> dp (delete (abs l) vars) (assignTrue l cnf)
  -- pure literal rule (positive)
  | ($v :: $vs, !((#(neg v) :: _) :: _)) -> dp vs (assignTrue v cnf)
  -- pure literal rule (negative)
  | ($v :: $vs, !((#v :: _) :: _)) -> dp vs (assignTrue (neg v) cnf)
  -- otherwise
  | ($v :: $vs, _) ->
    dp vs
       ((resolveOn v cnf) ++ (deleteClausesWith v (deleteClausesWith (neg v) cnf)
    ))
```

The first match clause states that the input formula has a solution when it is empty. The second match clause states 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 states 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 from the variable list and the clauses that include this literal from the formula. For example, $(p \vee q) \wedge (\neg p \vee r) \wedge (\neg p \vee \neg r)$ 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 states 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. The `resolveOn` function collects all pairs of clauses $p \vee C$ and $\neg p \vee D$ (let C and D be a disjunction of literals), and returns new clauses $C \vee D$.

The above definition of `dp` describes all rules of the Davis-Putnam algorithm 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. We can compare this implementation with the same algorithm implemented in OCaml in [46].

3.4.1.2 Pattern Matching for Resolution

We can elegantly define the `resolveOn` function using a sequential not-pattern.

First, let us show a naive implementation of `resolveOn`. The `resolveOn` function is defined with a single `matchAll` expression as follows.

```
def resolveOn v cnf := matchAll cnf as multiset (multiset integer) with
  | (#v :: $xs) :: (#(negate v) :: $ys) :: _
    -> unique (filter (\c -> not (tautology c)) (xs ++ ys))
```

The pattern for enumerating the pair of clauses $p \vee C$ and $\neg p \vee D$ is described simply utilizing pattern-matching for a multiset of multisets. Note that the above `resolveOn` removes tautological clauses by using the `tautology` predicate in the body of the match clause.

We can remove this use of the `tautology` predicate by using a sequential not-pattern discussed in Section 3.3.3. The sequential pattern is effectively used to describe that the literal `x` appeared in `xs` does not appear negatively in `ys`.

```
def resolveOn v cnf :=
  matchAll cnf as multiset (multiset integer) with
  | {(#v :: (@ & $xs)) :: (#(neg v) :: (@ & $ys)) :: _,
     !($l :: _, #(neg l) :: _)}
    ->  unique (xs ++ ys)
```

3.4.1.3 Separating Two Kinds of Loops

The SAT solver presented in this section is an important sample in the sense that it is the only sample that contains a loop that we cannot remove by pattern-match-oriented programming. This unremovable loop is the recursion of the `dp` function. This recursion is essential for narrowing the search tree. This narrowing is impossible by simple backtracking. On the other hand, all the other loops that can be implemented in backtracking algorithms are pushed into the patterns. In the traditional style, we usually describe the 1-literal rule and pure literal rules by combining several recursive functions such as `find`, `partition`, `subtract` and `intersect` [46]. Thus, pattern-match-oriented programming increases the readability of practical algorithms.

3.4.2 Graph Pattern Matching

This section demonstrates pattern matching for graphs as sets of edges and adjacency graphs, respectively.

3.4.2.1 Graphs as Sets of Edges

In this section, we pattern-match a graph as a set of edges. We can define a matcher and graph data as follows.

```
def graph := set edge
def edge := algebraicDataMatcher
  | edge integer integer

def graphData :=
  [ Edge 1 2, Edge 2 1, Edge 2 3,  Edge 2 4, Edge 3 4, Edge 4 5, Edge 4 6, Edge 4
     7
  , Edge 5 4, Edge 5 6, Edge 5 7,  Edge 6 4, Edge 6 5, Edge 6 7, Edge 7 4, Edge 7
     5
  , Edge 7 6, Edge 7 8, Edge 9 10, Edge 10 7 ]
```

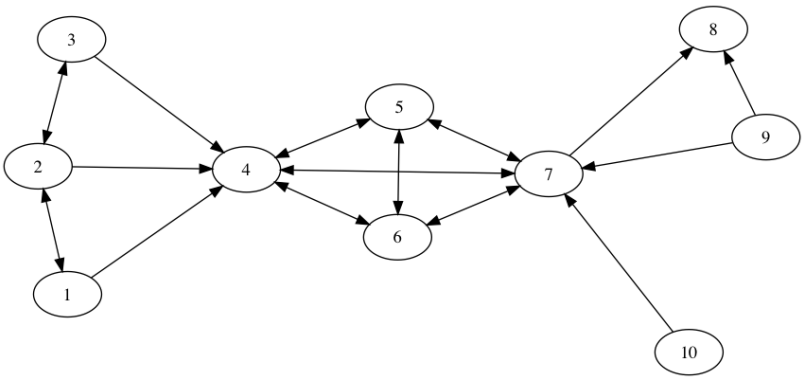

Figure 3.1: Visualization of `graphData`.

This sample graph is visualized in Section 3.1. This section lists several pattern-matching examples against this graph. Various patterns for graphs can be described using techniques introduced in this Chapter.

A pattern for listing all nodes that are accessible from `s` in two edges is described as follows.

```
let s := 1 in
  matchAll graphData as graph with
  | edge (#s & $x_1) $x_2 :: edge #x_2 $x_3 :: _
  -> x
-- [1, 3, 4, 5, 6, 7]
```

A pattern for listing all nodes that possess an edge from `s` but not to `s` is described utilizing a not-pattern effectively.

```
let s := 1 in
  matchAll graphData as graph with
  | edge #s $x :: !(edge #x #s :: _)
  -> x
-- [4]
```

A pattern for listing all routes from `s` to `e` is defined with a loop pattern. Egison allows users to use the `let` expression inside a pattern. The `let` expression is used to bind `s` to `$x_1`. Thanks to this `let`, we can describe elegantly an initial condition for `$x_1` for the loop pattern.

```
let (s, e) := (1, 8) in
  matchAll graphData as graph with
  | let x_1 := s in
      loop $i (2, $n)
           (edge #x_(i - 1) $x_i :: ...)
           (edge #x_(n - 1) (#e & $x_n) :: _)
  -> map (\i -> x_i) [1..n]
-- [[1, 4, 7, 8], ...]
```

A pattern for finding all cliques whose size is n is given in the following way. A double-nested loop pattern can be used for that purpose.

```
let n := 4 in
  matchAll graphData2 as graph with
  | edge $x_1 $x_2 :: loop $i (3, n)
                           (edge #x_1 $x_i :: loop $j (2, i - 1)
                                                      (edge #x_j #x_i :: ...)
                                                      ...)
                           _
  -> map (\i -> x_i) [1..n]
-- [[4, 5, 6, 7], ...]
```

In this section, we demonstrated pattern matching only for a directed graph. However, we can also define a matcher for undirected graphs using a matcher for edges that identifies `Edge a b` and `Edge b a`. The matcher for undirected edges is defined as a user-defined matcher, which we explained in Section 2.5.

3.4.2.2 Adjacency Graphs

This section demonstrates pattern matching for a weighted adjacency list. As shown in the program below, a matcher for a weighted adjacency list can be simply defined by composing matchers. `graphData` in the program below represents an airline network by a weighted adjacency list. The integers in `graphData` are the costs of time by hours to move between two cities.

```
def graph := multiset (string, multiset (string, integer))

def graphData :=
  [("Berlin", [("New York", 14), ("London", 2), ("Tokyo", 14), ("Vancouver", 13)
    ]),
   ("New York", [("Berlin", 14), ("London", 12), ("Tokyo", 18), ("Vancouver", 6)
    ]),
   ("London", [("Berlin", 2), ("New York", 12), ("Tokyo", 15), ("Vancouver", 10)
    ]),
   ("Tokyo", [("Berlin", 14), ("New York", 18), ("London", 15), ("Vancouver", 12)
    ]),
   ("Vancouver", [("Berlin", 13), ("New York", 6), ("London", 10), ("Tokyo", 12)
    ])]

-- List all routes that visit all cities exactly once and return to Berlin.
def trips :=
  let n := length graphData in
    matchAll graphData as graph with
    | (#"Berlin", (($s_1,$p_1) :: _)) ::
        loop $i (2, n - 1)
          ((#s_(i - 1), ($s_i, $p_i) :: _) :: ...)
          ((#s_(n - 1), (#"Berlin" & $s_n, $p_n) :: _) :: [])
    -> sum (map (\i -> p_i) [1..n]), map (\i -> s_i) [1..n]

head (sortBy (\(_, x), (_, y) -> compare x y)) trips)
-- (["London", "New York", "Vancouver", "Tokyo"," Berlin"], 46)
```

The above `matchAll` expression lists all routes from Berlin that visit all the cities exactly once and return to Berlin. This pattern can be used to solve the traveling salesman problem. A non-linear loop pattern is used to represent the pattern.

There are several graph database query languages [66, 72, 6]. The advantage of Egison over these query languages is its generality. Egison does not focus on pattern matching for graphs. Instead, Egison allows users to describe various patterns by just combining non-linear loop patterns and a small number of simple pattern constructors.

3.4.3 Computer Algebra

As an application of pattern-match-oriented programming, we have developed a computer algebra system [31]. We can implement a pattern-matching engine for mathematical expressions in a short program by defining a matcher for mathematical expressions.

```
def 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)]
    | sym $ as string with
      | Sym $name -> [name]
      | $ as something with
    | $tgt -> [tgt]
```

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

```
def 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 * (prod (map power y)) + (sum 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 systems in the field of which there are still notations that are popular among researchers of mathematics 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 the author. We have developed a natural method for importing tensor index notation into programming [32]. Thanks to this work, Egison became an appropriate computer algebra system for describing formulae of differential geometry. We will explain this feature in Chapter 5

3.4.4 Database Query Languages

Egison pattern matching can provide a unified query language for various kinds of databases. For example, let us consider a database of social network services and a query to list all users who are followed by the 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
  ((name #"Egison_Lang") & (id $uid) :: _,
   (fromID #uid) & (toID $fid) :: !((fromID #fid) & (toID #uid) :: _),
   (ID #fid) & (Name $fname) :: _) -> (fid, fname)
```

The above `matchAll` expression matches a tuple of the user table (`users`), the follow table (`follows`), and the user table. Each table is pattern-matched as a set. The second line pattern-matches the user table. `Name` and `ID` are pattern constructors

to access the column of a record. The pattern for the user table in the second line matches the user whose name is `"Egison_Lang"` and `$uid` is bound to the user ID of that user. The third line pattern-matches the follow table. `FromID` and `ToID` are pattern constructors to match the IDs of follower and followee. The user of `FromID` follows the user of `ToID`. IDs of the users who do not follow back the user whose ID is `uid` is pattern-matched using a not-pattern. The fourth line pattern-matches the user table again to get the name of the user whose ID is `fid` and returns the tuple `(fid, fname)`.

The conciseness of the queries is an important advantage of Egison over SQL [26]. 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-match-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)
```

List comprehensions also work as a sophisticated query language for relational databases [88]. The above query can be simply expressed also by list comprehensions. The advantage of Egison to list comprehensions as a query language is its generality. Egison can be used to express queries not only for relational databases but also for XML and graph databases. XML path language [7] and graph query languages [66, 72, 6] 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.

3.5 Related Work

This section compares our approach with list comprehensions (Section 3.5.1) and logic programming (Section 3.5.2) as other approaches to remove explicit recursions from programs.

3.5.1 List Comprehensions

List comprehensions [67] are another approach to hide explicit recursions. For example, list comprehensions also allow us to define `map` without explicit recursion:

```
map f xs = [ f x | x <- xs ]
```

However, list comprehensions are too specialized to enumerate elements of a list and do not allow us to describe complex enumerations as concisely as pattern-match-oriented programming. We can summarize the advantages of pattern-match-oriented programming against list comprehensions as follows:

1. Pattern-match-oriented programming requires less local variables;
2. Pattern-match-oriented programming is more expressive thanks to ad-hoc polymorphism of patterns and Egison specific pattern constructs such as loop patterns and sequential pattern;

3. Pattern matching can be used to describe conditional branches.

The second and third advantages are obvious. Therefore, in the rest of this section, we focus on the first advantage.

For illustrating the first advantage, we write a program that enumerates all the two combinations of elements in list comprehensions. We described the same program in pattern-match-oriented programming style in Section 3.3.2. `tails` is a function that returns all the suffixes of the argument list.

```
comb2 xs = [ (x,y) | x:ys <- tails xs, y <- ys ]
```

The variable `ys` is necessary for list comprehensions though it is unnecessary in pattern-match-oriented programming. Such variables that are necessary only in list comprehensions appear when the pattern is nested. As a result, nested join-cons patterns for lists (e.g. `unique` in Section 3.3.1.2) and nested cons patterns for multisets (e.g. an N-queen solver in Section 3.3.4.2 and a SAT solver in Section 3.4.1) cannot be described in list comprehensions as concisely as pattern-match-oriented programming.

3.5.2 Comparison with Logic Programming

Logic programming is a programming paradigm proposed with a similar motive of simplifying the descriptions of backtracking. Logic programming describes non-deterministic computations using unification instead of pattern matching. Unification is a more general notion compared with pattern matching. But the integration of non-determinism of logic programming and pattern matching is not obvious. For example, the pattern-matching facility of Prolog is specialized only for algebraic data types.

We can summarize the advantages of our approach against logic programming as follows:

1. Our approach proposes syntax specialized for pattern matching of non-free data types.

2. Our approach is compiler-friendly because our approach directly defines methods for decomposing data.

3. Our approach modularizes non-free data types by matchers.

3.5.2.1 Special Syntax for Pattern Matching of Non-free Data Types

We can describe non-linear patterns and backtracking in functional logic programming as we have shown in Section 2.2.1. However, logic programming languages do not provide a special syntactic construct for handling multiple pattern-matching results as the `matchAll` expression of our language. As a result, programs for non-free data types in logic programming are not as concise as ones in our approach. Curry [45] provides `findall` for handling multiple unification results. 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-match-oriented style is as follows.

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

The `findall` function takes a predicate (a function that returns a boolean value) and returns a list of values that satisfy the predicate. The predicate passed to

`findall` in the above code returns true when both `(_ ++ (x : _)) =:= xs` and `f x =:= y` return true. In Curry, `&` is a boolean operator and `=:=` represents an equational constraint. We can represent pattern matching using equational constraints. The equational constraint `(_ ++ (x : _)) =:= xs` binds the free variable `x` to an element of `xs`. Then, the equational constraint `f x =:= y` binds the variable `y` to `f x`. Finally, `findall` returns all the values bound to `y`. As a result, the above `findall` returns a list of results applying `f` to each element of `xs`.

3.5.2.2 Compiler-Friendly Pattern Definitions

The key difference between logic programming and our approach is in the method for defining pattern-match algorithms. Logic programming deduces the method for decomposing data from a method for constructing data. For example, in Curry, the cons pattern for multisets, whose name is `insert` in Curry, is defined as follows (we demonstrated this pattern constructor in Section 2.2).

```
1 insert x [] = [x]
2 insert x (y:ys) = x:y:ys ? y:(insert x ys)
```

The `insert` function is a non-deterministic function that returns the result of inserting the first argument to the second argument list. The infix operator `?` is called a non-deterministic operator. The `?` operator returns its second argument for example when the unification for the first argument fails. It is also possible to enumerate all the results of a non-deterministic function by using `findall` explained in Section 3.5.2.1. The first line defines how to insert `x` to the empty list. The second line defines how to insert `x` to a non-empty list. The first argument of the `?` operator is `x:y:ys` that means `x` is inserted into the head of the non-empty list. The second argument of the `?` operator is `y:(insert x ys)`. This part uses recursion for inserting the element `x` into the tail part of the argument list. `insert x ys` returns the non-deterministic results of inserting `x` to `ys`.

On the other hand, in our proposal, we define pattern constructors by describing how to decompose data directly. Thanks to that, in our approach, it is easy to control and optimize the execution process of the internal pattern-match process. For example, we can naturally define wildcard optimizations in our approach (Section 2.5.2).

3.5.2.3 Abstraction of Non-free Data Types by Matchers

In our approach, we modularize pattern-match methods not for each pattern but for each non-free data type using matchers. This abstraction of non-free data types improves the readability of programs by ad-hoc polymorphism of patterns as we explained in Section 2.3.3. Not only that, this abstraction also enables us to apply more complex optimization like pattern fusion that is explained in Section 2.5.3

3.6 Conclusion

In this chapter, we have 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. We listed many algorithms that can be represented more elegantly in 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 the potential to accelerate research of computer science as a whole. We hope our work leads to the further evolution of pattern matching and the future progress of data abstraction.

Chapter 4

Embedding Egison Pattern Matching into Haskell

In this chapter, we discuss methods for embedding Egison pattern matching into Haskell. There are three technical challenges for that: (1) we need to design typing rules for Egison pattern matching; (2) we need to convert Egison pattern-match expressions to a Haskell program; (3) we need to make converted Haskell programs type-inferable by GHC. In this chapter, we show our solution. We have implemented our solution in the Haskell library, *Sweet Egison*. This library has already been distributed via Hackage. Sweet Egison has three features that the original Egison interpreter does not have: (1) type errors are detected statically by the compiler; (2) execution speed is much faster; (3) the users can customize search strategies for pattern matching.

4.1 Overview

4.1.1 Usage of Sweet Egison

We start from the explanation of the `matchAll` expression of Sweet Egison. 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. A match clause is constructed using the quasi-quote [81] provided by Template Haskell [75]. 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 dfs [1, 2, 3] (List Something)
  [[mc| $x : $xs -> (x, xs) |]]
-- [(1, [2, 3])]
```

The `matchAll` expression of Sweet Egison takes an additional argument as its first argument. We specify a search strategy by the first argument. By default, we can choose the depth-first and breadth-first search strategies. In Sweet Egison, users can define search strategies by defining their own backtracking monads. It is an important feature of Sweet Egison when compared to the original Egison interpreter and Sweet Egison. We explain how to define backtracking monads in Section 4.3.2.

```
take 10 (matchAll dfs [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)]

take 10 (matchAll bfs [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)]
```

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

Matchers, the third argument of `matchAll`, play an important role in achieving ad-hoc polymorphism of patterns as the original Egison language. The matcher of the program below is `(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 change as follows.

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

The users can define matchers and pattern-match algorithms for each pattern of 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 for implementing this Haskell library. We explain how to define matchers in Sweet Egison in Section 4.3.4.

Matching in Sweet Egison allows non-linear patterns as the original Egison interpreter. The pattern that starts with `#` is 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 dfs [1, 5, 2, 4] (Multiset Eql)
  [[mc| $x : #(x + 1) : _ -> (x, x + 1) |]]
-- [(1, 2), (4, 5)]
```

The type of an expression inside a value pattern is inferable by the GHC type checker. Maintaining non-linear patterns in a static type system of Haskell is also a major technical challenge for implementing this Haskell library.

Sweet Egison also provides the `match` expression, which calculates only the first pattern-match result. The `match` expression is simply implemented as follows.

```
match s t m cs = head (matchAll s t 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.

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

poker :: [Card] -> String
poker cs = match dfs 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" |]]
```

Figure 4.1: Pattern matching for poker hand in Sweet Egison

We can use Sweet Egison for pattern matching of poker hand as shown in Figure 4.1. We can represent each poker hand in a single pattern by pattern matching a list of cards as a multiset. The `CardM` matcher is a matcher for playing cards. "`[`p_1`,` p_2`, ..., `p_n`]`" is syntactic sugar of a pattern whose form is "p_1 `:` p_2 `: ... :` p_n `: []`".

There are two features not implemented in Sweet Egison. First, some non-standard pattern constructs, such as loop patterns, indexed pattern variables, and sequential patterns, are not implemented in Sweet Egison. Loop patterns and indexed pattern variables are used for representing the repetitions inside a pattern (Section 3.2.6). Sequential patterns are used to control the order of pattern matching (Section 3.2.7). Second, the users of Sweet Egison cannot define pattern fusions. Sweet Egison cannot pattern-match patterns when defining a matcher. This is because the users define each pattern as a function. We explain this limitation in Section 4.3.6.

4.1.2 Compiling Method

Our key idea is to transform patterns into a program that uses backtracking monads such as the standard list monad. For example, this `matchAll` expression

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

is converted to the following Haskell program:

```
(\ (matcher, target) -> do
   (x, xs) <- cons matcher target -- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
   let (_, _) = consM matcher target -- (Something, Multiset Something)
   return (x, xs))
   (Multiset Something, [1, 2, 3]) -- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
```

The `cons` (the alias of `:`) and `consM` functions are user-defined functions that posses the pattern-match method for the cons pattern. They are called *next-target function* and *next-matcher function*, respectively. The `cons` function takes a matcher and a target and returns a list of results of decomposing the target. We need to pass a matcher to next-target functions to make them polymorphic. For example, the `cons` function for multisets returns all the pairs of an element and the rest of the target list.

```
cons (Multiset Something) [1, 2, 3]
-- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
```

The `consM` function defines how these decomposed data are pattern-matched in the next step.

```
consM (Multiset Something) [1, 2, 3]
-- (Something, Multiset Something)
```

Each result of the `cons` function (e.g., `1` and `[2, 3]`) is pattern-matched as `Something` and `Multiset Something`, respectively.

Next, we show the conversion of nested patterns. Patterns are transformed in sequence in the same `do` expression, therefore we can refer to the values that are bound to the pattern variables in the left side of a pattern. For example, the following non-linear pattern matching

```
matchAll dfs [1, 2, 3, 4] (Multiset Eql)
    [[mc| $x : #(x * 2) : _ -> (x, x * 2) |]]
-- [(1, 2), (2, 4)]
```

is converted as follows.

```
(\ (matcher, target) -> do
  (x, target') <- cons matcher target
   let (_, matcher') = consM matcher target
   (target'', _) <- cons matcher' target'
   let (matcher'', _) = consM matcher' target'
   value (x * 2) matcher'' target''
   return (x, y))
   (Multiset Eql, [1, 2, 3, 4])
```

In the above program, `value` is a next-target function for handling value patterns. The `value` function is user-defined like the `cons` function. Users can define how to handle value patterns for each non-free data type.

4.1.3 Organization of This Chapter

The rest of this chapter is organized as follows. Section 4.2 proposes a set of typing rules for `matchAll`, matchers, patterns, and match clauses. Section 4.3 explains the implementation of Sweet Egison. Section 4.5 shows the benchmarking results. Section 4.6 concludes the chapter.

4.2 Typing Rules

This section shows a set of typing rules that we designed for Typed Egison [54], 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 4.2 shows the set of typing rules. In the rules, meta-variables x, e, p, and C denotes a variable, expression, pattern, and 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 elements of type T. `Matcher` and `Pattern` are built-in type constructors: `Matcher` T is a type of a matcher for T; `Pattern` T 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

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

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

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

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

Figure 4.2: Typing rules for `matchAll` and patterns of Typed Egison

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.* This judgement denote that p has the type T under the type environment Γ and Δ updating Δ to Δ'. Δ is an additional type environment that describes the types of variables bound to the left of the pattern 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 (TP-Var) adds a type binding for each pattern variable. These assignments in Δ are used to type-check the content of a value pattern (TP-Value). The type environment Δ is updated and passed in a pattern in order from left to right (TP-Constructor).

In figure 4.2, we do not define the behavior for the case where a single pattern has multiple occurrences of the same pattern variables. 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 TP-Var. A value pattern `#x` refers to the value matched by the closest preceding pattern variable `$x`.

4.3 Implementation of Sweet Egison

Sweet Egison embeds Egison pattern matching into Haskell by converting patterns to a program that uses backtracking monads by utilizing the meta-programming facility of Haskell. There are three technical challenges for implementing Sweet Egison.

1. Prepare backtracking monads for both depth-first search and depth-first search.

*We borrow this notation from [51].

2. Design a set of translation rules that convert Egison pattern-match expressions to Haskell programs that are type-inferable.

3. Enable the users to define their own patterns and matchers.

This section explains our solution. Section 4.3.1 implements the backtracking monads for depth-first search and breadth-first search, and explains the method for defining new backtracking monads. Section 4.3.2 explains how Sweet Egison controls the search strategy for pattern matching utilizing these backtracking monads. Section 4.3.3 explains how we convert a pattern-match expression to a Haskell program that uses the backtracking monads. Section 4.3.4 explains how we define matchers in Sweet Egison. Section 4.3.5 explains the type of the pattern-match expressions in Sweet Egison. Section 4.3.6 explains the optimization techniques implemented in Sweet Egison.

4.3.1 Backtracking Monads

The goal of this section is to define backtracking monads that behave as follows.

```
toList (do
  x <- (fromList [1,2,3] :: DFS Integer)
  y <- fromList [1,2,3]
  return (x, y))
-- [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]

toList (do
  x <- (fromList [1,2,3] :: BFS Integer)
  y <- fromList [1,2,3]
  return (x, y))
-- [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(2,3),(3,2),(3,3)]
```

The first and second expressions use the `DFS` and `BFS` monads, respectively. We switch a search strategy by changing the monad. The `toList` and `fromList` functions define the conversion between the value of these backtracking monads and lists. To make these conversion functions polymorphic, we define the `MonadSearch` type class. The `MonadSearch` type class takes the type variable `m` as its argument. This argument is used to represent two class methods, `fromList` and `toList`, are polymorphic to `m`.

```
class MonadSearch m where
  fromList :: [a] -> m a
  toList   :: m a -> [a]
```

The implementation of the `DFS` monad is simple because we can use the list monad as the `DFS` monad. We define the type constructor `DFS` as the alias of the type constructor of lists.

```
type DFS = []
```

Then we can use the `id` function for both `fromList` and `toList`.

```
instance MonadSearch [] where
  fromList = id
  toList = id
```

Our implementation of the `BFS` monad follows Spivey's method [78]. It is known by Spivey that we can implement a monadic interface also for breadth-first search. Spivey's monad uses a list of lists as the data structure for a backtracking monad for breadth-first search. Here is an implementation of a monad for breadth-first search.

```
newtype BFS a = BFS [[a]]

instance MonadPlus BFS where
  mzero = BFS []
  mplus (BFS xss) (BFS yss) = BFS (merge xss yss)
   where
    merge :: [[a]] -> [[a]] -> [[a]]
    merge [] [] = []
    merge xss [] = xss
    merge [] yss = yss
    merge (xs:xss) (ys:yss) = (xs ++ ys) : merge xss yss

instance Monad BFS where
  return x = BFS [[x]]
  BFS [] >>= _ = BFS []
  BFS (xs:xss) >>= f = foldl mplus mzero (map f xs) mplus shift (BFS xss >>= f)
   where
    shift :: BFS a -> BFS a
    shift (BFS xss) = BFS ([] : xss)
```

In fact, the above program implements the identical algorithm that traverses the binary reduction tree we discussed in Section 2.4.2. The list of lists in this backtracking monad represents the list of lists of matching states in the same depth level in a binary reduction tree.

The conversion for a backtracking monad for breadth-first search is defined as follows.

```
instance MonadSearch BFS where
  fromList xs = BFS (map (\x -> [x]) xs)
  toList (BFS xss) = concat xss
```

We can investigate how this `BFS` monad backtracks in breadth-first order by the following code (the `negate` function changes the sign of the given integer).

```
do x <- (fromList [1..] :: BFS Integer)
   y <- fromList (map negate [1..])
   return (x, y))
--  [[(1, -1)], [(1, -2), (2, -1)], [(1, -3), (2, -2), (3, -1)], ...]
```

We have implemented the library that provides the above `DFS` and `BFS` monads. We make this program open to public as a Hackage library [35]. Sweet Egison uses this Haskell library.

4.3.2 Control of Search Strategy

For enabling the users to choose a search strategy, we transform pattern-match expressions to `do` expressions that use the monads that are instances of the `MonadSearch` type class. For example, the pattern-match expression

```
matchAll strategy [1..] (Set Something) [[mc| $x : $y : _ -> (x, y) |]]
```

is transformed as follows:

```
1 toList
2   ((\ (matcher, target) -> do
3        (x, target') <- fromList (cons matcher target)
4        let (_, matcher') = consM matcher target
5        (y, _) <- fromList (cons matcher' target')
6        let (_, _) = consM matcher' target'
7        return (x, y))
8      (strategy (Set Something, [1..])))
```

The `fromList` function is applied to the result of next-target functions. The `toList` function is applied to the return value of the `do` expression. The `strategy` function, the first argument of `matchAll`, is used to initialize the monad value that is passed to the `do` expression. The `strategy` function takes the tuple that consists of the matcher and the target of `matchAll`. As shown in Section 4.1.1, we pass `dfs` or `bfs` to the first argument of `matchAll`.

Now, we explain the implementation of `dfs` and `bfs`. The `dfs` and `bfs` functions take a single value and return the monad value. The `dfs` strategy is defined as follows.

```
dfs :: a -> DFS a
dfs x = [x]
```

The `bfs` strategy is defined as follows.

```
bfs :: a -> BFS a
bfs x = fromList [x]
```

We can use the `fromList` function for this purpose.

4.3.3 Translation Rules

Figure 4.3 shows the list of translation rules implemented in the `mc` quasi-quoter. The `mc` quasi-quoter converts a match clause to a function that takes a tuple of a matcher and target, and returns the results of evaluating the body for each pattern-match result. In Figure 4.3, $[mc|\ p\ \rightarrow\ e\ |]$ is abbreviated to $[|\ p\ \rightarrow\ e\ |]$. The rest of this section explains each translation rule by showing the specific examples.

4.3.3.1 Converting Wildcards

We do nothing to pattern-match a wildcard because pattern-matching of a wildcard always succeeds.

```
[mc| _ -> body |]
```

```
\ (_, _) -> return body
```

4.3.3.2 Converting Pattern Variables

The only difference from the translation rule of a wildcard is that we assign the target to the pattern variable. Pattern-matching of a pattern variable also always succeeds.

```
[mc| $x -> body |]
```

```
\ (_, target) -> do
  let x = target
  return body
```

4.3.3.3 Converting Value Patterns

A value pattern is converted to a program that calls the `value` next-target function. The `value` function checks whether its first and third arguments are equal or not. If they are equal, the `value` function returns a non-empty list. Otherwise, it returns the empty list.

$$
\begin{aligned}
&\text{WILDCARD}\\
&[|\ \_\ \rightarrow\ e\ |]\ =\ \backslash\ (\_,\_)\ \rightarrow\ return\ e\\
&\text{PATTERN VARIABLE}\\
&[|\ \$x\ \rightarrow\ e\ |]\ =\ \backslash\ (\_,t)\ \rightarrow\ let\ x\ =\ t\ in\ return\ e\\
&\text{VALUE PATTERN}\\
&[|\ \#v\ \rightarrow\ e\ |]\ =\ \backslash\ (m,t)\ \rightarrow\ fromList\ (value\ v\ m\ t)\ >>\ return\ e\\
&\text{EMPTY TUPLE PATTERN}\\
&[|\ ()\ \rightarrow\ e\ |]\ =\ \backslash\ ((),())\ \rightarrow\ return\ e\\
&\text{TUPLE PATTERN}\\
&[|\ (p_1,\cdots,p_n)\ \rightarrow\ e\ |]\ =\ \backslash\ ((m_1,\cdots,m_n),(t_1,\cdots,t_n))\ \rightarrow\\
&\qquad\qquad [|\ p_1\ \rightarrow\ [|\ (p_2,\cdots,p_n)\ \rightarrow\ e\ |]\\
&\qquad\qquad\quad (m_2,\cdots,m_n)\ (t_2,\cdots,t_n)\ |]\ m_1\ t_1\\
&\text{CONSTRUCTOR PATTERN}\\
&[|\ c\ p_1\ \cdots\ p_n\ \rightarrow\ e\ |]\ =\ \backslash\ (m,t)\ \rightarrow\ do\ \{\\
&\qquad\qquad (t_1,\cdots,t_n)\ \leftarrow\ \ fromList\ (c\ m\ t);\\
&\qquad\qquad let\ (m_1,\cdots,m_n)\ =\ cM\ m\ t;\\
&\qquad\qquad [|\ (p_1,\cdots,p_n)\ \rightarrow\ e\ |]\ (m_1,\cdots,m_n)\ (t_1,\cdots,t_n)\ \}
\end{aligned}
$$

Figure 4.3: Translation rules of match clauses

```
[mc| #v -> body |]
```

```
\ (matcher, target) -> do
    fromList (value v matcher target)
    return body
```

4.3.3.4 Converting Tuple Patterns

When the pattern is the empty-tuple pattern, the corresponding matcher and target must be the empty tuple. Otherwise, the pattern-match expression causes a type error at compile time.

```
[mc| () -> body |]
```

```
\ ((), ()) -> return body
```

The elements of a non-empty tuple pattern are converted in sequence from the pattern of the first element in order. We use recursion in the translation rule of the tuple pattern for implementing this sequential conversion.

```
[mc| ($x, $y) -> body |]
```

```
\ ((m1, m2), (t1, t2)) -> do
  let x = t1
  do let y = t1
     return body
```

4.3.3.5 Converting Constructor Patterns

A constructor pattern is converted to a program that calls the corresponding next-target and next-matcher functions. The number of elements of the tuples returned by these functions is the same as the number of the argument of constructor patterns. We use the translation rules for tuple patterns for handling the next targets and matchers.

```
[mc| cons $x $y -> body |]
```

```
\ (matcher, target) -> do
    (t1, t2) <- fromList (cons matcher target)
    (m1, m2) <- consM matcher target
    do let x = t1
       do let y = t2
          return body
```

4.3.4 Defining Matchers

This section explains how we define user-defined patterns and matchers.

The relationship between a matcher and its target datatype is represented using a multi-parameter type class [48] that declares no class method. The type constraint `Matcher m t` asserts that `m` is a matcher for `t`.

```
class Matcher m t
```

We can declare that data of some type are matchers for data of another 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
```

4.3.4.1 Matchers for Collections

This section explains how we define polymorphic patterns in Sweet Egison by showing the definition of the cons pattern for a multiset.

The `List m` matcher is defined as a data type whose type and data constructor are identical.

```
data List m = List m
```

The `List m` matcher is an instance of the `Matcher` type class. This instance declaration asserts that `List m` is a matcher for a list of elements of the type `t` when `m` is a matcher for the type `t`.

```
instance Matcher m t => Matcher (List m) [t]
```

The definition of the `Multiset` datatype is completely the same as that of `List`.

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

The patterns for collections are defined as functions of the `CollectionPat` type class. In the following code, the definitions of the `nil` and `join` patterns are omitted.

```
class CollectionPattern m t where
  type ElemM m
  type ElemT t
  consM :: m -> t -> (ElemM m, m)
  cons ::  m -> t -> (ElemT t, t)
```

Index type families (`ElemM` and `ElemT`) are used for extracting the matcher and type of the elements of a target collection. The `consM` function takes a matcher and returns the next matchers. The `cons` function takes a target datum and returns the next targets. 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 $x _`.

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

```
instance Matcher m t => CollectionPattern (List m) [t] where
  type ElemM (List m) = m
  type ElemT [t] = t
  consM (List m) _ = (m, List m)
  cons _        []       = []
  cons (List _) (x : xs) = [(x, xs)]

instance Matcher m t => CollectionPattern (Multiset m) [t] where
  type ElemM (Multiset m) = m
  type ElemT [t] = t
  consM (Multiset m) _ = (m, Multiset m)
  cons (Multiset _) xs = matchAll dfs xs (List Something)
    [[mc| $hs ++ $x : $ts -> (x, hs ++ ts) |]]
```

4.3.4.2 The `Eql` Matcher

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

First, we start from the definition of the `Eql` matcher. `Eql` is a matcher for data types that belong the `Eq` type class.

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

As we explained in Section 4.3.3, the value pattern is converted to the application of the `value` next-target function. The `value` next-target function is defined as a function of the `ValuePattern` type class to achieve ad-hoc polymorphism. The first argument of `value` is a value passed to the value pattern. The `value` function compares this value with the target that is passed as the third argument. The `value` function has the default definition that compares the value and the target using `==`.

```
class Eq t => ValuePattern m t where
  value :: t -> m -> t -> [()]
  default value :: Eq t => t -> m -> t -> [()]
  value e _ v = if e == v then [()] else []
```

We use the default definition of `value` for the `Eql` matcher. The `Eql` matcher uses `==` for pattern-matching value patterns.

```
instance Eq a => ValuePattern Eql a
```

Ad-hoc polymorphism of `value` allows us to define equality of non-free data types for which we cannot use `==`. For example, we can define a pattern-match method for a value pattern of a multiset as follows.

```
instance (Eq a, Matcher m a, ValuePattern m a)
  => ValuePattern (Multiset m) [a] where
  value e () (Multiset m) v = match dfs (xs, ys) (List m, Multiset m)
    [ [mc| ([], []) -> True |]
    , [mc| ($x : $xs, #x : #xs) -> True |]
    , [mc| _ -> False |] ]
```

4.3.4.3 The `Something` Matcher

The definition of the `Something` matcher is simple in Sweet Egison. This is because the pattern-match method for wildcards and pattern variables are already defined in our translation rules. In the following code, we just declare that we can use the `Something` matcher for all the data types.

```
data Something = Something

instance Matcher Something a
```

4.3.5 Typing the Pattern-Match Expressions

This section explains the type of the `matchAll` expression of Sweet Egison. The `matchAll` expression is defined in Haskell as follows.

```
matchAll
  :: (Matcher m t, MonadSearch s)
  => ((m, t) -> s (m, t))
  -> t
  -> m
  -> [(m, t) -> s r]
  -> [r]
matchAll strategy target matcher =
  concatMap (\clause -> toList (strategy (matcher, target) >>= clause))
```

The type constraint `Matcher m t` states that `m` is a matcher for `t`. The first argument is a search strategy whose type is `((m, t) -> s (m, t))`. As we explained in Section 4.3.2, a search strategy is a function that creates data of a type that is an instance of the `MonadSearch` type class. The type of the forth argument is `[(m, t) -> s r]`. This represents the type of a list of converted match clauses. As we explained in Section 4.3.3, a match clause is converted to a function that takes a tuple of a matcher and target and returns the results evaluating the body for each pattern-match result.

4.3.6 Optimization

We have implemented wildcard optimization and pattern fusion in Sweet Egison. This section explains how we have implemented them.

4.3.6.1 Wildcard Optimization

Wildcard optimization omits the calculation of the target that matches wildcard (Section 2.5.2). For example, if the second argument of the cons pattern is a wildcard, we do not need to calculate the rest.

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

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

For wildcard optimization, we add an argument to the definition of patterns to tell whether the argument pattern is wildcard or not. If an argument is a wildcard, the pattern returns `undefined` as the next target. Thus, we omit the calculation.

```
cons (GP, WC) (Multiset Something) [1, 2, 3]
-- [(1, undefined), (2, undefined), (3, undefined)]
cons (GP, GP) (Multiset Something) [1, 2, 3]
-- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
```

`GP` and `WC` are data constructors of `PP` and abbreviations of a general pattern and wildcard, respectively. `PP` is the abbreviation of a pattern for patterns.

```
data PP = WC | GP
```

Let us see how the definition of the cons pattern for multisets changes. The `cons` function takes a pair of patterns for patterns as its first argument because the cons pattern takes two patterns as its argument.

```
class CollectionPattern m t where
  ...
  cons :: (PP, PP) -> m -> t -> (ElemT t, t)
  ...
```

Here is a definition of the `cons` function. When the second argument of the cons pattern is a wildcard, the `cons` function does not compute the rest elements.

```
instance Matcher m t => CollectionPattern (Multiset m) [t] where
  ...
  cons (_, WC) (Multiset _) xs = map (\x -> (x, undefined)) xs
  cons _       (Multiset _) xs = matchAll dfs xs (List Something)
    [[mc| $hs ++ $x : $ts -> (x, hs ++ ts) |]]
  ...
```

4.3.6.2 Pattern Fusions

Pattern fusion is invented for applying wildcard optimization for nested patterns (Section 2.5.3). For example, we can apply pattern fusion for a join pattern whose second argument is a cons pattern.

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

If we handle join and cons separately, we calculate all the suffixes and get the first element of each suffix.

```
do (_, target') <- join (Wildcard, GeneralPat) target
   (x, _) <- cons (GeneralPat, Wildcard) target'
   return x
```

On the contrary, if we handle join and cons at the same time and apply wildcard optimization, we can omit the calculation of all the suffixes of the target list and just enumerate its elements.

```
do (_, x, _) <- joinCons (Wildcard, GeneralPat, Wildcard) target
   return x
```

We define the `joinCons` function as follow.

```
joinCons (WildCard, GeneralPat, Wildcard) (List Something) [1, 2, 3]
-- [(undefined, 1, undefined), (undefined, 2, undefined), (undefined, 3,
    undefined)]
```

Currently, Sweet Egison does not allow the users to define new pattern fusion. The transformation from `_ ++ $x : _` to `joinCons _ $x _` is hard-coded in the implementation of the `mc` quasi-quoter that transforms patterns to do expressions. On the other hand, the original Egison language has a user interface for defining pattern fusion. Pattern fusion is defined by pattern-matching patterns. We may borrow this user interface when we implement Sweet Egison as a GHC extension.

```
list a = matcher
    | _ ++ $ : _ as a with
      | tgt -> tgt
    | _ ++ $ as list a with
      | tgt -> tails tgt
    | $ ++ $ as (list a, list a) with
      ...
```

4.3.7 Summary

Sweet Egison transforms an Egison pattern-match expression to a Haskell program whose appearance is similar to a program that we manually write when describing backtracking. We summarize our method for this transformation as follows.

1. The users can define their own search strategies by defining a backtracking monad (Section 4.3.1).
2. Sweet Egison transforms a match clause to a program that uses a backtracking monad by the quasi-quote of Template Haskell (Section 4.3.2 and Section 4.3.3).
3. Sweet Egison allows the users to define polymorphic patterns as the original Egison interpreter (Section 4.3.4).
4. Both wildcard optimization and pattern fusion are implemented in Sweet Egison (Section 4.3.6).

4.4 Related Work

Over the decades, several pattern-match extensions have been proposed and implemented in Haskell. Views [92] 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 [39] 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 [89] provides the user-customizable pattern matching with backtracking. However, first-class patterns do not support non-linear patterns. The pattern-match system of Egison 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

	comb2 ($n = 15,000$)	comb2 ($n = 30,000$)	perm2 ($n = 5,000$)	perm2 ($n = 10,000$)
Functional style	0.323	1.264	0.717	3.862
miniEgison	16.123	63.45	6.287	25.133
Sweet Egison (w/o pattern fusion)	1.291	5.077	0.686	3.572
Sweet Egison	0.291	1.035	0.690	3.483

Table 4.1: Execution time of `comb2` and `perm2` in seconds

above pattern-match extensions are available as open-source and compilable 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 [69].

Sweet Egison is not the first attempt to implement the Egison pattern-match facility as a library of mainstream functional programming languages. Before Sweet Egison, we developed miniEgison, a library that provides the Egison pattern-match facility, in Scheme [34] and Haskell.

We briefly explain the approach taken by miniEgison. MiniEgison also transforms Egison pattern-match expressions to the Haskell programs. But the method for this transformation is different. MiniEgison embeds an interpreter for Egison pattern matching, whereas Sweet Egison embeds a compiler. MiniEgison calls this interpreter via the `patternMatch` function that implements the internal pattern-match algorithm, reduction of matching states that have a stack of matching atoms, explained in Section 2.4. The `patternMatch` function takes a pattern, target, and matcher and returns pattern-match results. Here is an example of the conversion by MiniEgison.

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

```
map (\x, y) -> (x, y))
    (patternMatch [1, 2, 3] (Multiset Something) ($x : $y : _))
```

In implementing miniEgison, we heavily reuse the ideas from the implementation of the first-class patterns [69]: a pattern is defined as a function that takes a target and returns a pattern-match result, which is represented by a heterogeneous list. The development of the conversion of miniEgison is not straightforward because we need to make converted programs automatically type-inferable by GHC. Especially, typing of matching states was challenging. GHC extensions such as GADTs and existential types play essential roles for this purpose.

4.5 Performance

This section discusses the execution performance of Sweet Egison. First, we compared the performance of programs that enumerate two combinations and permutations of elements, namely `comb2` and `perm2`, implemented in a traditional functional style and pattern-match-oriented style in Haskell. Table 4.1 shows the benchmark results. The definitions of the `comb2` and `perm2` functions are shown in Figure 4.4. The `comb2` function contains a join-cons pattern that we can apply pattern fusion. Let us remind you that pattern fusion is not implemented in miniEgison.

	CDCL (20 vars)	CDCL (50 vars)	CDCL (75 vars)	CDCL (100 vars)
Egison	2.114	40.565	1175.28	n/a
miniEgison	0.045	1.459	15.957	347.82
Sweet Egison	0.026	0.120	1.036	27.471

Table 4.2: Execution time of SAT solvers in seconds

```
comb2 :: Int -> [(Int, Int)]
comb2 n = matchAll dfs [1 .. n] (List Something)
  [[mc| _ ++ $x : _ ++ $y : _ -> (x, y) |]]

comb2Native :: Int -> [(Int, Int)]
comb2Native n = [ (x, y) | (x : ts) <- tails [1 .. n], y <- ts ]

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

perm2Native :: Int -> [(Int, Int)]
perm2Native n = go [1 .. n] [] []
 where
  go [] _ acc = acc
  go (x : xs) rest acc =
    [ (x, y) | y <- rest ++ xs ] ++ go xs (rest ++ [x]) acc
```

Figure 4.4: Benchmark programs

We also benchmarked our implementation of conflict-driven clause learning (CDCL), an algorithm for solving SAT, in a pattern-match-oriented programming style. Table 4.2 shows the benchmark results. The complete benchmarking program is accessible on GitHub [56]. Our program for CDCL does not contain a pattern that we can apply pattern fusion.

We compiled our benchmarks by the Glasgow Haskell Compiler (GHC) 8.10.7 and ran them on a 2.3GHz Dual-Core Intel Core i5 processor with 16GB of RAM. We passed the options `-O3 -threaded -rtsopts -with-rtsopts=-N` to GHC for compiling the above programs. The benchmark results show that:

- Pattern-match-oriented style programs written in Sweet Egison are as fast as traditional functional style programs in Haskell.
- Sweet Egison runs about 100 times faster than the original Egison interpreter.
- Sweet Egison runs about 10 times faster than miniEgison.
- Pattern fusion is important for improving execution performance.

The performance difference between miniEgison and Sweet Egison comes from miniEgison's internal pattern-match algorithm as a reduction of matching states, which makes it hard to be optimized by GHC.

4.6 Conclusion

In this chapter, we showed that user-customizable non-linear pattern matching with backtracking implemented in the Egison programming language is implementable as a Haskell library. 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.

The proposed library makes Egison pattern matching 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 Sweet Egison 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.

Chapter 5

Computer Algebra System and Tensor Index Notation

5.1 Overview

In mathematics, many notations have been invented for the concise representation of mathematical formulae. Tensor index notation is one of such notations and has been playing a crucial role in describing formulae in mathematical physics. In this chapter, we show a programming language that can deal with symbolic tensor indices by introducing a set of tensor index rules that is compatible with two types of parameters, i.e., scalar and tensor parameters. When a tensor parameter obtains a tensor as an argument, the function treats the tensor argument as a whole. In contrast, when a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor. On a language with scalar and tensor parameters, we can design a set of index reduction rules that allows users to use tensor index notation for arbitrary user-defined functions without requiring additional description. Furthermore, we can also design index completion rules that allow users to define the operators concisely for differential forms such as the wedge product, exterior derivative, and Hodge star operator. In our proposal, all these tensor operators are user-defined functions and can be passed as arguments of high-order functions. We have implemented our proposal in the computer algebra system that is implemented in the interpreter of our language using Sweet Egison.

This chapter is organized as follows. Section 5.2 introduces related work. Section 5.3 proposes our symbolic index reduction rules for importing tensor index notaion. Section 5.4 proposes our symbolic index completion rules for describing formulae that handle differential forms. Section 5.5 demonstrates our computer algebra system. Section 5.6 evaluates our proposal. Section 5.7 concludes this chapter.

5.2 Related Work

This section classifies the existing method for handling tensor index notation in programs.

5.2.1 Syntax-based Approach

The current major method for dealing with tensors is using a special syntax for describing loops for generating multi-dimensional arrays. The `Table` [13] expression from the Wolfram language is such a syntax construct. $X_{ij} + Y_{ij}$ is represented as follows with the `Table` expression. The following program assumes

that the length of each dimension corresponding to each index of the tensor are a constant `M`.

```
Table[X[[i,j]] + Y[[i,j]],{i,M},{j,M}]
```

For contracting tensors, we use the `Sum` [12] expression inside `Table`. ${X^i}_k {Y^k}_j$ is represented as follows.

```
Table[Sum[X[[i,k]] * Y[[k,j]],
          {k,M}],
      {i,M},{j,M}]
```

This method has the advantage that we can use an arbitrary function defined for scalar values for tensor operations. The following Wolfram program represents $\partial X_{ij}/\partial x^k$. `D` is the differential function in the Wolfram language.

```
Table[D[X[[i,j]],x[[k]]],{i,M},{j,M},{k,M}]
```

Due to this advantage, the Wolfram language has been used by mathematicians in actual research [61, 60]. However, in this method, we cannot modularize tensor operators such as tensor multiplication by functions. Due to this restriction, we cannot syntactically distinguish applications of different tensor operators, such as tensor multiplication, wedge product, and Lie derivative, in programs. This is because we need to represent these tensor operators combining `Table` and `Sum` every time when we use them. Modularization by functions is also important for combining index notation with high-order functions. If we can pass tensor operators to high-order functions, we can represent a formula like "$X_{i_1} X_{i_2} ... X_{i_n}$" (the number of tensors multiplied depends on the parameter n) by passing the operator for tensor multiplication to the fold function [22]. There are other existing works that take the same approach with the Wolfram language. NumPy's `einsum` operator [11], Diderot's EIN operator [55], and tensor comprehensions [91] are such work. Some of them provide a syntactic construct whose appearance is similar to mathematical formulae. However, they have the same restriction on function modularization. This restriction comes from the requirement that users need to specify the indices of the result tensors (e.g., "$_{ij}$" of $A_{ij} = X_{ik} Y_{kj}$) for determining whether to contract a pair of identical symbolic indices.

5.2.2 Library-based Approach

Another method for dealing with tensors is using library functions prepared for handling symbolic tensor indices. Using this method enables index notation to be directly represented in a program as the mathematical expressions. However, in this method, it is not easy for many users to define new tensor operators by themselves. One of the reasons is that index rules differ for each operator as mentioned in Introduction. We need to describe the index rules for each tensor operator when defining them. For example, the program below is the definition of tensor addition in SageMath [15].

```
def __add__(self, other):
    permutation = list(range(other._tensor.tensor_rank()))
    for other_index in range(other._tensor.tensor_type()[0]):
        if other._con[other_index] == self._con[other_index]:
            permutation[other_index] = other_index
        else:
            permutation[other_index] = self._con.index(other._con[other_index])
    for other_index in range(other._tensor.tensor_type()[1]):
        if other._cov[other_index] == self._cov[other_index]:
            permutation[other._tensor.tensor_type()[0] + other_index]\
```

```
                = other._tensor.tensor_type()[0] + other_index
        else:
            permutation[other._tensor.tensor_type()[0] + other_index]\
                = other._tensor.tensor_type()[0] + self._cov.index(other._cov[
    other_index])

    result = self.__pos__()
    result._tensor = result._tensor + other.permute_indices(permutation)._tensor
    return result
```

The C++ library by Åhlander [16], ITensor [42] library in Julia, and SageMath [15] also provides the tensor operators that can handle symbolic tensor indices in a similar way.

We propose a method for reducing these additional descriptions for handling symbolic tensor indices for defining tensor operators. By simplifying index rules, we can unify index rules of many tensor operators. As a result, we can set the default index rules and omit the descriptions of index rules from the definitions of tensor operators.

5.2.3 Array-Oriented Programming

Array-oriented programming languages, such as APL and J, take a completely different approach. They do not use tensor index notation for representing tensor calculus. Instead, they invented a new notion, *function rank* [21]. Function rank specifies how to map the operators to the components of tensors. When the specified function rank is 0 for an argument matrix, the operator is mapped to each scalar component of the matrix $(A_i + B_{jk})$. When the specified function rank is 1 for an argument matrix, the operator is mapped to the rows of the matrix by regarding the matrix as a vector of vectors $(A_j + B_{ij})$. When the specified function rank is 2 for an argument matrix, the operator is applied to the matrix directly $(A_i + B_{ij})$.

```
J> (2 $ 1 2) +"1 0 (2 2 $ 10 20  30 40)
11 12
21 22

31 32
41 42
J> (2 $ 1 2) +"1 1 (2 2 $ 10 20  30 40)
11 22
31 42
J> (2 $ 1 2) +"1 2 (2 2 $ 10 20  30 40)
11 21
32 42
```

Function ranks allow users to apply functions defined for scalar values to tensors. A similar idea to the function rank is also imported into various programming languages and frameworks, including Wolfram [8] and NumPy [14]. However, the function rank has a limitation that it does not allow to represent an expression that requires transposition of an argument tensor: e.g., $A_{ij} + B_{ji}$ (this expression requires the transposition of the matrix B).

We show that our simplified index rules are compatible with scalar and tensor parameters, which is a simplified notion of function rank. Our simplified index rules allow us to set the default index rules. The combination of the default index rules and function ranks allows us to apply functions defined for scalar values to tensors using index notation.

5.2.4 Differential Forms

Differential forms are important concepts in differential geometry. Differential forms are used to describe formulae in differential geometry in a coordinate independent way. By combining the operators for differential forms, such as the wedge product and exterior derivative, we can concisely describe various formulae in differential geometry. Differential forms can be represented as tensors and formulae that we can describe using differential forms include formulae in tensor calculus [74].

There are numerous independent works [53, 79, 15] for programming differential forms. What makes programming of differential forms difficult is that differential forms are alternating multi-linear forms (e.g., $dx \wedge dy = -dy \wedge dx$) and have many representation. These works prepare special data structures for representing this property of differential forms. As a result, programming for tensor calculus and differential forms have been studied separately.

Our approach represents differential forms using multi-dimensional arrays as we do for tensors. This approach allows us to use tensor index notation for defining the operators for differential forms. We show that our simplified index rules are useful for defining the operators not only for tensors but also differential forms. By designing the index completion rules for omitted indices properly, we become able to concisely define the operators for the differential forms. We introduce our method for programming of differential forms in Section 5.4.

5.2.5 Symbolic Tensor Calculus

There are two types of approaches for manipulating tensors: the first approach aims to compute each component of tensors and represents a tensor using some data structures such as arrays; the second approach aims to simplify formulae that contains tensors with symbolic indices and handles a tensor as a whole as a symbol. The existing methods that we introduced so far take the first approach and aims to compute each component of tensors. We also takes the first approach.

There are several libraries that implement the second approach. For example, Ricci of the Wolfram language [59] and `itensor` of Maxima [9, 87, 77] are such libraries. These libraries implement rewrite rules, such as $g^{ij}\Gamma_{jkl} = \Gamma^{i}{}_{kl}$ where g^{ij} is a metric tensor and $d(\omega \wedge \eta) = d\omega \wedge \eta + (-1)^p \omega \wedge d\eta$ where ω is a p-form, for simplifying formulae that contains tensors with symbolic indices.

The behaviors and definitions of tensor operators in these two approach are completely different. We focuse on the first approach.

5.3 Index Reduction Rules for Importing Tensor Index Notation

This section presents a new method for importing tensor index notation into programming languages. Briefly, it is achieved by introducing two types of parameters, *scalar parameters* and *tensor parameters*, and simple index reduction rules. First, we introduce scalar and tensor parameters. Second, we introduce a set of index reduction rules that is compatible with them. The combination of scalar and tensor parameters and the proposed index reduction rules enables us to apply user-defined functions to tensors using tensor index notation. We demonstrate our proposal using the code in the Egison programming language. Egison has a similar syntax to the Haskell programming language.

```
def min $x $y := if x < y then x else y
```

(a) Definition of the `min` function

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_j) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\ min(2,10) & min(2,20) & min(2,30)\\ min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ij} = \begin{pmatrix}1 & 1 & 1\\ 2 & 2 & 2\\ 3 & 3 & 3\end{pmatrix}_{ij}$$

(b) Application of the `min` function to the vectors with different indices

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_i) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\ min(2,10) & min(2,20) & min(2,30)\\ min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ii} = \begin{pmatrix}min(1,10)\\ min(2,20)\\ min(3,30)\end{pmatrix}_i = \begin{pmatrix}1\\2\\3\end{pmatrix}_i$$

(c) Application of the `min` function to the vectors with identical indices

Figure 5.1: Definition and application of `min` function

5.3.1 Scalar and Tensor Parameters

Scalar and tensor parameters are a similar notion to the function rank [21]. When a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor. In contrast, when a tensor parameter obtains a tensor as an argument, the function treats the tensor argument as a whole. We call a function that takes only scalar parameters *scalar function* and a function that takes only tensor parameters *tensor function*. For example, "`+`", "`-`", "`*`", and "`/`" should be defined as scalar functions; a function for multiplying tensors and a function for matrix determinant should be defined as tensor functions. The difference between function rank and these two types of parameters is that function rank allows users to control the level of mapping, whereas scalar and tensor parameters only allow users to specify whether we map the function to each component of the tensor or not. Instead, the proposed tensor index reduction rules combined with scalar and tensor parameters allow users to control the level of mapping.

Figure 5.1a shows the definition of the `min` function as an example of a scalar function. The `min` function takes two numbers as the arguments and returns the smaller one. "`$`" is prepended to the beginning of the parameters of the `min` function. It means the parameters of the `min` function are scalar parameters. When a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor like tensor product as figures 5.1b and 5.1c. As Figure 5.1c, if the indices of the tensors given as arguments are identical, the result matrix is reduced to the vector that consists of the diagonal components. This reduction is defined in the proposed tensor index reduction rules that will be explained in the next section. Thus the `min` function can handle tensors even though it is defined without considering tensors.

Figure 5.2a shows the definition of the "." function as an example of a tensor function. "." is a function for multiplying tensors. "`%`" is prepended to the beginning of the parameters of the "." function. It means the parameters of the "." function are tensor parameters. As with ordinary functions, when a tensor is provided to a tensor parameter, the function treats the tensor argument as a whole maintaining its indices. "." is defined as an infix operator. For defining an infix operator, we enclose the name of a function with parenthesis.

```
def (.) %t1 %t2 := contractWith (+) (t1 * t2)
```

(a) Definition of the "." function

$$\begin{pmatrix}1\\2\\3\end{pmatrix}^{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contractWith(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}^{i}_{i}) = 10 + 40 + 90 = 140$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contractWith(+, \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}) = \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{j} = contractWith(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}) = \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}$$

(b) Application of the "." function

Figure 5.2: Definition and application of "." function

$$R^{i}_{\ jkl} = \frac{\partial \Gamma^{i}_{\ jl}}{\partial x^{k}} - \frac{\partial \Gamma^{i}_{\ jk}}{\partial x^{l}} + \Gamma^{m}_{\ jl}\Gamma^{i}_{\ mk} - \Gamma^{m}_{\ jk}\Gamma^{i}_{\ ml}$$

(a) Formula of Riemann curvature tensor

```
R=Table[D[Γ[[i,j,l]],x[[k]]] - D[Γ[[i,j,k]],x[[l]]]
       +Sum[Γ[[m,j,l]] Γ[[i,m,k]]
          - Γ[[m,j,k]] Γ[[i,m,l]],
            {m,M}],
        {i,M},{j,M},{k,M},{l,M}]
```

(b) Wolfram program that represents the formula in Figure 5.3a

```
def R~i_j_k_l := withSymbols [m]
  ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l
```

(c) A program by the proposed language that represents the formula in Figure 5.3a

Figure 5.3: Representations of Riemann curvature tensor formula

In Figure 5.2a, "`+`" and "`*`" are scalar functions for addition and multiplication, respectively. `contractWith` is a function to contract a tensor that has pairs of a superscript and subscript with identical symbols. Figure 5.2b shows the example for calculating the inner product of two vectors using the "." function. We can use the "." function for any kind of tensor multiplication such as tensor product and matrix multiplication as well as inner product.

Let us show another example in differential geometry. When the mathematical expression in Fugure 5.3a is expressed in a standard way in the Wolfram language, it becomes a program such as the one shown in Figure 5.3b. The same expression can be expressed in our system as shown in Figure 5.3c. Our system supports two types of indices, both superscripts and subscripts. A subscript is represented by "`_`". A superscript is represented by "`~`". A double loop consisting of the `Table` and `Sum` expressions appears in the program in the Wolfram language, whereas the program in our system is flat, similar to the mathematical expression. This is achieved by using tensor index notation in the program. In particular, the reason that the loop structure by the `Sum` expression in the Wolfram language does not appear in our expression to express $\Gamma^{m}_{\ jk}\Gamma^{i}_{\ ml} - \Gamma^{m}_{\ jl}\Gamma^{i}_{\ mk}$ is that the "." function

can handle Einstein summation notation. It is emphisized that the program `(∂/∂ Γ~i_j_k x~l)` in this example expresses $\frac{\partial \Gamma^i{}_{jk}}{\partial x^l}$ in the first term on the right-hand side. In the Wolfram language, the differential function `D` is applied to each tensor component, but our differential function `∂/∂` is applied directly to the tensors.

The differential function `∂/∂` is defined in a program as a scalar function like the `min` function. When a tensor is provided as an argument to a scalar function, the function is applied automatically to each component of the tensor. Therefore, when defining a scalar function, it is sufficient to consider only a scalar as its argument. That is, in the definition of the `∂/∂` function, the programmers need write the program only for the case in which the argument is a scalar value. Despite that, the program `∂/∂ Γ~i_j_k x~l` returns a fourth-order tensor. Thus, we can import tensor index notation including Einstein notation into programming if we clearly distinguish between tensor functions such as "." and scalar functions such as "`+`" and `∂/∂`.

5.3.2 Reduction Rules for Tensors with Indices

This section presents a whole set of index reduction rules that are compatible with the scalar and tensor parameters. Let us consider the reduction rules only for a single tensor, and that is enough to define the set of the index reduction rules. This is because the reduction rules are applied only after a scalar function is applied to tensor arguments as seen in Figure 5.1.

5.3.2.1 Multiple Identical Symbolic Indices

Tensors are reduced only when they have identical symbols as their indices. First, let us discuss the cases that tensors have identical superscripts or subscripts.

When multiple indices of the same symbol appear, our system converts it to the tensor composed of diagonal components for these indices. After this conversion, the leftmost index symbol remains. For example, the indices "`_i_j_i`" convert to "`_i_j`". In our system, unbound variables are treated as symbols. These symbols can be used as indices of tensors. In our system, a tensor is expressed by enclosing its components with "`[|`" and "`|]`". We express a higher-order tensor by nesting this notation, as we do for an n-dimensional array. In this thesis, we show the evaluation result of a program in the comment that follows the program. "`--`" is the inline comment delimiter of our language.

```
[|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]_i_j --
    [|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]_i_j

[|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]_i_i -- [|11,22,33|]_i

[|[|[|1,2|],[|3,4|]|],[|[|5,6|],[|7,8|]|]|]_i_j_i -- [|[|1,3|],[|6,8|]|]_i_j
```

When three or more subscripts of the same symbol appear, our system converts it to the tensor composed of diagonal components for all these indices.

```
[|[|[|1,2|],[|3,4|]|],[|[|5,6|],[|7,8|]|]|]_i_i_i -- [|1,8|]_i
```

Superscripts and subscripts behave symmetrically. When only superscripts are used, they behave in exactly the same manner as when only subscripts are used.

```
[|[|[|1,2|],[|3,4|]|],[|[|5,6|],[|7,8|]|]|]~i~j~i -- [|[|1,3|],[|6,8|]|]~i~j
```

```
E({A, xs}) =
  if e(xs) = [] then
    {A, xs})
  elsif e(xs) = [{k,j}, ...] & p(k,xs) = p(j,xs) then
    E({diag(k, j, A), remove(j, xs))
  elsif e(xs) = [{k,j}, ...] & p(k,xs) != p(j,xs) then
    E({diag(k, j, A), update(k, 0, remove(j, xs)))
```

Figure 5.4: Pseudo code of index reduction

5.3.2.2 Superscripts and Subscripts with Identical Symbols

Next, let us consider a case in which the same symbols are used for a superscript and a subscript. In this case, some existing work [16] automatically contracts the tensor using "+". In contrast, our system just converts it to the tensor composed of diagonal components, as in the above case. However, in this case, the summarized indices become a *supersubscript*, which is represented by "~_".

```
[|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]~i_i -- [|11,22,33|]~_i
```

Even when three or more indices of the same symbol appear that contain both superscripts and subscripts, our system converts it to the tensor composed of diagonal components for all these indices.

```
[|[|[|1,2|],[|3,4|]|],[|[|5,6|],[|7,8|]|]|]~i~i_i -- [|1,8|]~_i
```

The reason not to contract it immediately is to parameterize an operator for contraction. The components of supersubscripts can be contracted by using the `contractWith` function. The `contractWith` function receives a function to be used for contraction as the first argument, and a target tensor as the second argument. This feature allows us to implement a tensor multiplication function.

```
contractWith (+) [|11,22,33|]~_i -- 66
```

In the above program, the "`+`" function is passed to `contractWith` as a prefix operator. When an infix operator is enclosed with parenthesis, it becomes an prefix operator.

The `contractWith` function is defined using the `contract` built-in function. The `contract` function takes a tensor and returns the list that consists of diagonal components.

```
contract [|11,22,33|]~_i -- [11,22,33]
```

5.3.2.3 Pseudo Code for Index Reduction

Figure 5.4 shows the pseudo-code of index reduction explained in the above. `E(A,xs)` is a function for reducing a tensor with indices, where `A` is an array that consists of tensor components and `xs` is a list of indices appended to `A`. For example, `E(A,xs)` works as follows with the tensor whose indices are "`~i_j_i`". We use `1`, `-1`, and `0` to represent a superscript, subscript, and supersubscript, respectively.

```
E({[|[|[|1,2|],[|3,4|]|]
    ,[|[|5,6|],[|7,8|]|]|]
 ,[{i,1}, {j,-1}, {i,-1}]}) =
{[|[|1,3|],[|6,8|]|], [{i,0}, {j,-1}]}
```

Let us explain the helper functions used in Figure 5.4: `e(xs)` is a function for finding pairs of identical indices from `xs`; `diag(k, j, A)` is a function for creating the tensor that consists of diagonal components of `A` for the `k`-th and `j`-th order; `remove(k, xs)` is a function for removing the `k`-th element from `xs`; `p(k, xs)` is a function for obtaining the value of the `k`-th element of the assoc list `xs`; `update(k , p, xs)` is a function for updating the value of the `k`-th element of the assoc list `xs` to `p`. These functions work as follows.

```
e([{i, 1}, {j, -1}, {i, -1}])       = [{1,3}]
diag(1, 2, [|[|11,12|],[|21,22|]|]) = [|11,22|]
p(2, [{i, 1}, {j, -1}])             = -1
remove(2, [{i, 1}, {j, -1}])        = [{i, 1}]
update(2, 0, [{i, 1}, {j, -1}])     = [{i, 1}, {j, 0}]
```

5.3.3 Implementation of Scalar and Tensor Parameters

Let us explain how to implement scalar and tensor parameters. The implementation of tensor parameters is the same with the ordinary parameters because the function treats the argument tensor as it is. In contrast, a function with scalar parameters is converted to a function only with tensor parameters by using the `tensorMap` function as follows.

```
\ $x $y -> ...
-- => \ %x %y ->
--      tensorMap (\ %x -> tensorMap (\ %y -> ...) y))
--                x
```

As the name implies, the `tensorMap` function applies the function of the first argument to each component of the tensor provided as the second argument. When the result of applying the function of the first argument to each component of the tensor provided as the second argument is a tensor with indices, it moves those indices to the end of the tensor that is the result of evaluating the `tensorMap` function.

Let us review the `min` function defined in Figure 5.1a as an example. This `min` function can handle tensors as arguments as follows.

```
min [|1,2,3|]_i [|10,20,30|]_j -- [|[|1,1,1|],[|2,2,2|],[|3,3,3|]|]_i_j
min [|1,2,3|]_i [|10,20,30|]_i -- [|1,2,3|]_i
```

Note that the tensor indices of the first evaluated result are "`_i_j`". If the `tensorMap` function simply applies the function to each component of the tensor, the result of this program is `[|[|1 1 1|]_j [|2 2 2|]_j [|3 3 3|]_j|]_i`. However, as explained above, if the results of applying the function to each component of the tensor are tensors with indices, it moves those indices to the end of the tensor that is the result of evaluating the `tensorMap` function. This is the reason that the indices of the evaluated result are "`_i_j`". This mechanism enables us to directly apply scalar functions to tensor arguments using index notation as the above example.

Next, let us review the "." function defined in Figure 5.2a as an example of a tensor function. When a tensor with indices is given as an argument of a tensor function, it is passed to the tensor function maintaining its indices. It allows us to directly apply tensor functions to tensor arguments using index notation as in the following example.

```
[|1,2,3|]~i . [|10,20,30|]_i -- 140
[|1,2,3|]_i . [|10,20,30|]_j -- [|[|10,20,30|],[|20,40,60|],[|30,60,90|]|]_i_j
[|1,2,3|]_i . [|10,20,30|]_i -- [|10,40,90|]_i
```

$$\partial/\partial \begin{pmatrix} f \\ g \end{pmatrix}_i \begin{pmatrix} x \\ y \end{pmatrix}^j = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}_{ij}$$

Figure 5.5: Index rule for partial derivative

Tensor parameters are rarely used compared with scalar parameters because a tensor parameter is used only when defining a function that contracts tensors.

5.3.4 Inverted Scalar Arguments

The ∂/∂ function, the derivative operator, in Figure 5.3c is a scalar function. However, ∂/∂ is not a normal scalar function. The ∂/∂ function is a scalar function that inverts indices of the tensor given as its second argument as shown in Figure 5.5. For example, the program `(∂/∂ Γ~i_j_k x~l)` returns the fourth-order tensor with the indices "`~i_j_k_l`".

To define scalar functions such as ∂/∂, we use *inverted scalar parameters*. Inverted scalar parameters are represented by "`*$`". A program that uses inverted scalar parameters is transformed as follows. The `flipIndices` function is a primitive function for inverting the indices of a tensor provided as an argument upside down. Supersubscripts remain as supersubscripts even if they are inverted.

```
def ∂/∂ $f *$x := ...
-- => def ∂/∂ %f %x := tensorMap (\ %f -> tensorMap (\ %x -> ...) (flipIndices x
    ))
--                            f
```

In the following example, we apply ∂/∂ to tensors.

```
∂/∂ [|(r * (sin θ)),(r * (cos θ))|]_i [|r,θ|]_j
-- [|[|(sin θ),(r * (cos θ))|],[|(cos θ),(-1 * r * (sin θ))|]|]_i~j
∂/∂ [|(r * (sin θ)),(r * (cos θ))|]_i [|r,θ|]_i
-- [|(sin θ),(-1 * r * (sin θ))|]~_i
```

5.3.5 The `withSymbols` Expression

The `withSymbols` expression is syntax for generating new local symbols as the `Module` [10] expression in the Wolfram language. One-character symbols that are often used as indices of tensors such as `i`, `j`, and `k` are often used in another part of a program. Generating local symbols using `withSymbols` expressions enables us to avoid variable conflicts for such symbols.

The `withSymbols` expression takes a list of symbols as its first argument. These symbols are valid only in the expression given in the second argument of the `withSymbols` expression.

```
withSymbols [i] contractWith (+) ([|1,2,3|]~i * [|10,20,30|]_i) -- 60
```

It acts in a special way when the evaluation result of the body of the `withSymbols` expression contains the symbols generated by the `withSymbols` expressions. In that case, the result tensor is transposed to shift those symbols backward and remove them. In the following evaluation result, the matrix is transposed because `j` is shifted backward before it removed. This mechanism is useful to handle differential forms that will be discussed in the next section.

```
withSymbols [j] [|[|1,2|],[|3,4|]|]_j_i) -- [|[|1,3|],[|2,4|]|]_i
```

5.3.6 Tensor Declaration

In our system, when binding a tensor to a variable, we can specify the type of indices in the variable name. For example, we can bind different tensors to `g__`, `g~~`, `R____`, and `R~___`. This feature is also implemented in Maxima [9] and simplifies variable names. In Figure 5.3c, we bind a tensor to the variable with symbolic indices `R~i_j_k_l`. It is automatically desugared as follows.

```
def R~i_j_k_l := ...
-- => def R~___ := withSymbols [i,j,k,l]
--      (transpose [i,j,k,l] ...)
```

This syntactic sugar renders a program closer to the mathematical expression. The `transpose` function is a built-in function for transposing the tensor in the second argument as specified in the first argument.

5.3.7 Declaring Symmetric and Antisymmetric Tensors

Symmetric and anti-symmetric tensors often appear in differential geometry. We can use symmetry to skip descriptions and calculation of symmetric components. This section designs a syntax construct for declaring symmetry and anti-symmetry of a tensor.

There is no special notation for declaring symmetry of tensors. For example, the symmetries of Riemann curvature tensors are declared using equations as follows.

$$R_{abcd} = -R_{bacd}$$

$$R_{abcd} = -R_{abdc}$$

$$R_{abcd} = R_{cdab}$$

We thought that it is redundant to declare tensor symmetries using the above equation in a program. First, we need a complicated program to determine which parts of the tensor are symmetric from the above equations. Second, the symmetry declaration independent of the definition of a tensor itself makes the specification of programming languages complex.

For these reasons, we want to define a tensor and declare its symmetries at the same time. We found a hint in the mathematical notation for the commutator. This notation is used to represent the sum and difference between values combining two elements enclosed by parenthesis.

$$[a, b] = ab - ba$$

$$\{a, b\} = ab + ba$$

This notation is also used in tensor indices.

$$R_{\{ab\}cd} = R_{abcd} + R_{bacd}$$

$$R_{[ab]cd} = R_{abcd} - R_{bacd}$$

Using this notation, we can describe the first two above symmetries of Riemann curvature tensor as follows.

$$R_{\{ab\}cd} = 0$$

$$R_{ab\{cd\}} = 0$$

By extending this notation, we can also describe the third symmetry.

$$R_{[(ab)(cd)]} = 0$$

We noticed that by borrowing this notation we can merge the tensor definition and symmetry declaration in a single expression. We show an example using the definition of R_{abcd}.

$$R_{abcd} = g_{ai}R^{i}{}_{bcd}$$

We declare symmetries of R_{abcd} by adding parenthesis in the list of symbolic indices in the left-side of equation.

$$R_{[\{ab\}\{cd\}]} = g_{ai}R^{i}{}_{bcd}$$

We explain how we implemented this notation. We desugar the following tensor declaration

```
def X{_i_j} := ...
```

as follows.

```
def X_i_j :=
 let tmpX_i_j := ... in
   generateTensor
     (\i j -> if i > j then tmpX_j_i
              else          tmpX_i_j)
     (tensorShape tmpX_#_#)
```

The desugared program uses the `generateTensor` expression. Thanks to the call-by-need strategy, we do not compute all the components of `tmpX`. We can apply the same method for anti-symmetric tensors.

```
def X[_i_j] := ...
```

The above tensor declaration is transformed as follows.

```
def X_i_j :=
 let tmpX_i_j := ... in
   generateTensor
     (\i j -> if i > j then -tmpX_j_i
              else if i = j then 0
              else           tmpX_i_j)
     (tensorShape tmpX_#_#)
```

Finally, we show a case that symmetry declaration reduces the description of a program. We found this case when writing a program for actual research of differential geometry. We define a tensor $\bar{R}_{abcd}$ that possesses the same symmetry as Riemann curvature tensor. The tensor $\bar{R}_{abcd}$ is defined as follows.

$$\bar{R}_{00cd} = 0, \bar{R}_{0b0d} = -p^2 g_{bd}, \bar{R}_{0bcd} = -p\nabla_b J_{cd},$$

$$\bar{R}_{abcd} = R_{abcd} - p^2 J_{bc}J_{ad} + p^2 J_{ac}J_{bd} + 2p^2 J_{ab}J_{cd}$$

This tensor is defined using the `generateTensor` expression as follow.

```
def R'_i_j_k_l :=
  generateTensor
    (\is -> match is as list integer with
       | [#1, #1, _, _] -> 0
       | [_, _, #1, #1] -> 0
       | [#1, $b, #1, $d] -> -1 * p^2 * g_(b - 1)_(d - 1)
```

```
      | [$a, #1, #1, $d] ->      p^2 * g_(a - 1)_(d - 1)
      | [#1, $b, $c, #1] ->      p^2 * g_(b - 1)_(c - 1)
      | [$a, #1, $c, #1] -> -1 * p^2 * g_(a - 1)_(c - 1)
      | [#1, $b, $c, $d] -> -1 * p * ∇J_(b - 1)_(c - 1)_(d - 1)
      | [$a, #1, $c, $d] ->      p * ∇J_(a - 1)_(c - 1)_(d - 1)
      | [$a, $b, #1, $d] -> -1 * p * ∇J_(d - 1)_(a - 1)_(b - 1)
      | [$a, $b, $c, #1] ->      p * ∇J_(c - 1)_(a - 1)_(b - 1)
      | [$a, $b, $c, $d] -> R_(a - 1)_(b - 1)_(c - 1)_(d - 1)
                            + -1 * p^2 * J_(b - 1)_(c - 1) * J_(a - 1)_(d - 1)
                            +      p^2 * J_(a - 1)_(c - 1) * J_(b - 1)_(d - 1)
                            +  2 * p^2 * J_(a - 1)_(b - 1) * J_(c - 1)_(d - 1))
    [5, 5, 5, 5]
```

By using our notation for declaring symmetry, we can reduce the number of branches in the `generateTensor`.

```
def R'{[_i_j][_k_l]} :=
  generateTensor
    (\is -> match is as list integer with
      | [#1, #1, _, _] -> 0
      | [#1, $b, #1, $d] -> -1 * p^2 * g_(b - 1)_(d - 1)
      | [#1, $b, $c, $d] -> -1 * p * ∇J_(b - 1)_(c - 1)_(d - 1)
      | [$a, $b, $c, $d] -> R_(a - 1)_(b - 1)_(c - 1)_(d - 1)
                            + -1 * p^2 * J_(b - 1)_(c - 1) * J_(a - 1)_(d - 1)
                            +      p^2 * J_(a - 1)_(c - 1) * J_(b - 1)_(d - 1)
                            +  2 * p^2 * J_(a - 1)_(b - 1) * J_(c - 1)_(d - 1))
    [5, 5, 5, 5]
```

5.3.8 Pattern Matching for Tensor Indices

For defining a tensor operator, we sometimes need to rearrange symbolic tensor indices of the argument tensors. For example, here is a mathematical formulae that defines the covariant derivative operator.

$$\begin{aligned}\nabla_c T^{a_1\cdots a_r}{}_{b_1\cdots b_k} = &\frac{\partial}{\partial x^c} T^{a_1\cdots a_r}{}_{b_1\cdots b_k} \\ &+ \Gamma^{a_1}{}_{dc} T^{da_2\cdots a_r}{}_{b_1\cdots b_k} + \cdots + \Gamma^{a_r}{}_{dc} T^{a_1\cdots a_{r-1}d}{}_{b_1\cdots b_k} \\ &- \Gamma^{d}{}_{a_1c} T^{a_1\cdots a_r}{}_{db_2\cdots b_k} - \cdots - \Gamma^{d}{}_{b_kc} T^{a_1\cdots a_r}{}_{b_1\cdots b_{k-1}d}\end{aligned}$$

The symbolic tensor indices appended to the tensor T in left-side and right-side of the equation are different.

In order to define such an operator, we allow to pattern-match symbolic tensor indices of the argument tensors.

```
def ∇_c T~(a_1)...~(a_r)_(b_1)..._(b_k) :=
  ∂/∂ T~(a_1)...~(a_r)_(b_1)..._(b_k) x~c
  + sum (map (\i -> Γ~(a_i)_d_c .
                    T~(a_1)...~(a_(i-1))~d~(a_(i+1))...~(a_r)_(b_1)..._(b_k))
             [1..r])
  - sum (map (\i -> Γ~d_(b_i)_c .
                    T~(a_1)...~(a_r)_(b_1)..._(b_(i-1))_d_(b_(i+1))..._(b_k))
             [1..k])
```

We allow to use "..." as we do in mathematical formulae. A pattern for tensor indices that contains "..." matches the logest possible part.

5.4 Index Completion Rules for Tensors with Omitted Indices

By designing the index completion rules for omitted indices properly, we can extend the proposed method explained so far to express a calculation handling

the differential forms.

5.4.1 Representation of Differential Forms

This section explains how we represent differential forms in programs. In our method, n-forms are represented by rank-n tensors. Let us show examples in 3-dimensional Euclidean space. One-forms dx, dy, and dz are represented as $\begin{pmatrix}1 & 0 & 0\end{pmatrix}$, $\begin{pmatrix}0 & 1 & 0\end{pmatrix}$, and $\begin{pmatrix}0 & 0 & 1\end{pmatrix}$, respectively. Two-forms $dx \wedge dy$ and $dz \wedge dx$ are represented as $\begin{pmatrix}0 & 1 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix}$ and $\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 1 & 0 & 0\end{pmatrix}$, respectively.

In mathematics, n-forms can be regarded as a function that takes n tangent vectors and returns a value. This return value is the result of multiplying the tensor that represents the differential form and the tangent vectors. Especially, the result of applying the tangent vector v to the one-form ω is inner product:

$$\omega(v) = \lfloor \omega \rfloor_i v^i$$

where $\lfloor \omega \rfloor$ is the tensor representation of ω. Generally, the result of applying the tangent vectors $v_1, \cdots, v_n$ to the n-form ω is determined by the following formula.

$$\omega(v_1, \cdots, v_n) = \left(\frac{1}{n!} \sum_{\sigma \in S_n} sgn(\sigma) \lfloor \omega \rfloor_{i_{\sigma_1} \cdots i_{\sigma_n}} \right) (v_1)^{i_1} \cdots (v_n)^{i_n}$$

We call $\frac{1}{n!} \sum_{\sigma \in S_n} sgn(\sigma) \lfloor \omega \rfloor_{i_{\sigma_1} \cdots i_{\sigma_n}}$ the *normal representation* of the differential form ω. We use the `antisymmetrize` function for calculating the normal representation of differential forms. For example, the normal representation of $dx \wedge dy$ is calculated as follows.

```
antisymmetrize [| [| 0, 1, 0 |], [| 0, 0, 0 |], [| 0, 0, 0 |] |]
-- [| [| 0, 1 / 2, 0 |], [| -1 / 2, 0, 0 |], [| 0, 0, 0 |] |]
```

5.4.2 Index Completion Rules

We distinguish differential forms from the other tensors in programs by omitting indices. In mathematics, we denote a p-form by appending only $(n - p)$ indices to an n-th order tensor. For example, the third order tensor $\omega^i{}_j$ denotes a matrix-valued one-form. We can use this notation for representing tensor-valued differential forms in programs.

This section shows the design of index completion rules for omitted indices that allows us to concisely define the operators for differential forms. Let `A` and `B` be scalar-valued 2-forms. For general scalar functions, we complement the same indices to each argument tensor as follows.

```
A + B -- => A_t1_t2 + B_t1_t2
```

We know that the above completion is natural from the following example.

$$\lfloor dx \wedge dy + dy \wedge dz \rfloor_{t_1 t_2} = \lfloor dx \wedge dy \rfloor_{t_1 t_2} + \lfloor dy \wedge dz \rfloor_{t_1 t_2}$$

In most of the cases, if we complement indices as above, we can represent the operation for differential forms. However, this completion is not suitable for functions specially defined for differential forms such as the wedge product and exterior derivative. In the case of the wedge product, we would like to append the different indices to each argument as follows.

```
A ∧ B -- => A_t1_t2 . B_t3_t4
```

For example, the following formula holds in four-dimensional Euclidean space.

$$\lfloor (dx \wedge dy) \wedge (dz \wedge dw) \rfloor_{t_1 t_2 t_3 t_4} = \lfloor dx \wedge dy \rfloor_{t_1 t_2} \lfloor dz \wedge dw \rfloor_{t_3 t_4}$$

We introduce the "`!`" operator for this purpose. If the "`!`" operator is prepended to function application, the omitted indices are complemented by the latter method. With this method, the function for calculating the wedge product is defined in one line as follows.

```
def (∧) %A %B := A !. B
```

We can also define the exterior derivative in one line as follows.

```
def d %A := !(flip ∂/∂) coord A
```

The `flip` function swaps the arguments of a two-argument function. It is used to transpose the result. We set the symbols for representing the coordinates to `coord`. We can use the wedge product and exterior derivative defined above as follows.

```
def coord := [| x, y, z |]

d x -- [| 1, 0, 0 |]
d x ∧ d y -- [| [| 0, 1, 0 |], [| 0, 0, 0 |], [| 0, 0, 0 |] |]
d z ∧ d x -- [| [| 0, 0, 0 |], [| 0, 0, 0 |], [| 1, 0, 0 |] |]
```

Next, we define Hodge star operator. We use the following mathematical formula for Hodge star operator [73].

$$*A = \sqrt{\det |g|} \cdot \epsilon_{i_1 \ldots i_n} \cdot A_{j_1 \ldots j_k} \cdot g^{i_1 j_1} \cdots g^{i_k j_k} \cdot e^{i_{k+1}} \wedge \ldots \wedge e^{i_n} \tag{5.1}$$

We can directly translate the above formula as follows.

```
def hodge %A :=
  let k := dfOrder A in
    withSymbols [i, j]
      (sqrt (abs (det g_#_#))) * (foldl (.) ((ϵ N k)_(i_1)..._(i_N) . A..._(j_1)
    ..._(j_k))
                                                  (map (\t -> g~(i_t)~(j_t)) [1..k]))
```

In the above program, the `dfOrder` function returns p when it obtains an p-form; the `det` function returns the determinant of the argument matrix; ϵ is the Levi-Civita symbols; "`#`" used in the indices as `g_#_#` represents a dummy symbol. All instances of "`#`" are treated as different symbols. In a program that deals with high-order tensors, the number of symbols used for indices increases. A dummy symbol suppresses that. This mechanism makes it easier to distinguish indices, thereby also improves the readability of the program.

5.5 Demonstration

This section shows that the proposed system is practical by showing a program of a certain size. By modifying the programs in this section, we can use our language for actual research. Figure 5.6b shows the mathematical formulae of Christoffel symbols and curvature form. These formulae appear in the 4th, 5th, and 8th programs in Figure 5.6a. We can see the sample programs that use the functions defined above in Egison Mathematics Notebook [31].

```
In [1]: def x := [| θ, φ |]

In [2]: def g_i_j := [| [| r^2, 0 |], [| 0, r^2 * (sin θ)^2 |] |]_i_j

In [3]: def g~i~j := [| [| 1 / r^2, 0 |], [| 0, 1 / (r^2 * (sin θ)^2) |] |]~i~j

In [4]: def Γ_i_j_k := (1 / 2) * (∂/∂ g_i_k x~j + ∂/∂ g_i_j x~k - ∂/∂ g_j_k x~i)

In [5]: def Γ~i_j_k := withSymbols [m] g~i~m . Γ_m_j_k

In [6]: def ω~i_j := Γ~i_j_#

In [7]: def d %t := !(flip ∂/∂) x t

In [8]: def Ω~i_j := withSymbols [k]
          antisymmetrize (d ω~i_j + ω~i_k ∧ ω~k_j)

In [9]: Ω~1_1
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

```
In [10]: Ω~1_2
```

$$\begin{pmatrix} 0 & \frac{sin(\theta)^2}{2} \\ \frac{-sin(\theta)^2}{2} & 0 \end{pmatrix}$$

```
In [11]: Ω~2_1
```

$$\begin{pmatrix} 0 & \frac{-1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}$$

```
In [12]: Ω~2_2
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

(a) A program for calculating Riemann curvature tensor of S^2 using the formula of curvature form

$$\Gamma_{ijk} = \frac{1}{2}\left(\frac{\partial g_{ij}}{\partial x^k} + \frac{\partial g_{ik}}{\partial x^j} - \frac{\partial g_{kj}}{\partial x^i}\right) \qquad \Gamma^i{}_{jk} = g^{im}\Gamma_{mjk} \qquad \Omega^i{}_j = d\omega^i{}_j + \omega^i{}_k \wedge \omega^k{}_j$$

(b) Mathematical formulae used in the above program

Figure 5.6: Program for differential geometry in our language

5.6 Evaluation

In this section, we evaluate how simple our definitions of tensor operators are by comparing to the other works. Table 5.1 shows the numbers of lines for defining tensor operators in ITensor [42], Karczmarczuk's Haskell library [53], SageMath [15], and our work. We chose these libraries for comparison because their definitions of tensor operators are compact compared to the other existing work. To make the comparison fair, we do not count the number of lines for error handling. The other approaches require additional descriptions for error handling because they do not relax the index rules like us.

The definitions of the tensor operators by our work is shortest among the existing work. The reason why our definitions are short is that our simplified index rules allow us to remove the descriptions of index rules from the definitions of tensor operators. Especially, the numbers of lines required to define tensor

Number of lines	ITensor (Julia)	Karczmarczuk	SageMath	Egison
addition	5	n/a	27	0
multiplication	17	n/a	27	1
derivative	n/a	n/a	n/a	0
wedge product	n/a	8	9	1
exterior derivative	n/a	9	8	1
hodge star	n/a	8	14	5

Table 5.1: The number of lines for defining operators for tensors and differential forms

addition and derivative operators are zero. It is achieved by defining the addition and derivative operators as scalar functions. The definition of tensor addition by ITensor is also short.

```
function (A::ITensor + B::ITensor)
  C = copy(A)
  C .+= B
  return C
end
```

This is because the broadcast operation, which specifies how to map the operation to the components of tensors, is defined in ITensor. When the dot mark is prepended to an operator like the line 3 of the above definition, the operator is broadcasted. In our approach, scalar functions are automatically mapped to each tensor component without any actions by users. The language-level support of scalar parameters allows us to define the arithmetic operators for tensor even shorter.

The definitions of the operators for differential forms by our work is also shortest among the existing work. This is because the reuse of our simplified index rules releases us from describing complex algebraic rules for differential forms. Especially, we can define the wedge product and exterior derivative in one line. It is an important discovery that we can reuse symbolic index rules for tensors for defining operators for differential forms. In addition, representing differential forms using tensors enables us to define operators for differential forms using index notation. It allows us to translate the formula of the hodge operator directly to a program as shown in Section 5.4.2.

Finally, let us note that the derivative operator ∂/∂ for tensors is supported only by our proposal. For this purpose we discussed to flip the superscript and subscript in programs to allow a vector as the second argument of the derivative operator as explained in Section 5.3.4. The derivative operator that supports symbolic tensor indices widen the range of formulae we can describe in programs using index notation. For example, users cannot describe a program that calculate Riemann curvature tensors without the derivative operator using index notation as shown in Figure 5.3c. In the existing library-based approaches, the function for calculating Riemann curvature tensors is provided as a library function.

5.7 Conclusion

In this chapter, we showed that we can import tensor index notation into a programming language by introducing a set of symbolic index reduction rules that is compatible with scalar and tensor parameters. The proposed method allows users to define functions that handle symbolic tensor indices without additional

descriptions for handling these symbolic tensor indices. We demonstrated the proposed method by showing our proof-of-concept programming language Egison. The proposed method can be implemented also in the other programming languages. It is an interesting research topic to think about how to incorporate the proposed method into existing programming languages. For example, we think it is possible to introduce the concept of scalar and tensor parameters using a static type system [68]. In a language with a static type system, whether the parameter of a function is a scalar or tensor parameter can be specified when specifying the type of the function.

In particular, it is of substantial significance to incorporate the proposed method into programming languages such as Formura [64] and Diderot [55] that have a compiler to generate code for executing tensor calculation efficiently. For example, incorporating the proposed method into Formura would enable us to describe physical simulation using not only the Cartesian coordinate system but also more general coordinate systems such as the polar and spherical coordinate systems in simple programs.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have proposed several syntax constructs and facilities of programming languages. We have demonstrated that these syntax constructs and facilities widen the range of algorithms that we can describe concisely. Specifically, the contributions of the thesis are as follows.

- In Chapter 2, we presented the design of our pattern-match system for non-free data types. We proposed three criteria that must be fulfilled for practical pattern matching for non-free data types: (1) efficiency of the backtracking algorithm for non-linear patterns, (2) user-customizability of pattern-match algorithm, and (3) polymorphism in patterns. We showed how our language design naturally arises from our criteria of practical pattern matching.
- In Chapter 3, we advocated a new programming paradigm called pattern-match-oriented programming. We introduced many programming techniques that replace explicit recursions with an intuitive pattern by confining recursions inside patterns. We also proposed several non-standard pattern constructs, such as not-patterns, loop patterns, and sequential patterns, derived from our pattern-match facility. We classified these techniques as pattern-match-oriented programming design patterns.
- In Chapter 4, we presented the implementation of a Haskell library that embed our pattern-match system into Haskell. We proposed a method for transforming Egison pattern-match expressions to an equivalent Haskell program that uses a backtracking monad. We also designed a set of typing rules for Egison pattern matching. We showed benchmark results that confirm our library can be used in practical situations.
- In Chapter 5, we discussed a method for importing notations from tensor calculus into programming languages. We proposed a set of index reduction and completion rules that allows us to define many tensor operators without descriptions for handling symbolic tensor indices. In addition to that, we presented a new syntax construct for declaring symmetries and anti-symmetries of a tensor. We represented many formulae in differential geometry by a program whose appearance is very close to the formulae.

6.2 The Three Steps for Creating New Syntax Constructs

While developing new features of programming languages, we become aware of the following steps for creating new syntax constructs.

Step 1. Discover a gap between our recognition and description of an algorithm.

Step 2. Design new syntax for describing algorithms in a form close to our recognition.

Step 3. Design a method for executing the new syntax.

In this section, we categorize our contributions in each step. It gives us insight for answering research questions about notations: "Is there a general way to find a new notation?", "Is there a formal method for measuring the superiority of multiple notations?"

6.2.1 Step 1: Discover a Gap between Our Recognition and Description of an Algorithm

We have noticed the cognitive gaps four times for pattern matching.

1. When we defined the `mapWithBothSides` function, we noticed the importance of pattern matching with multiple results (Section 1.2).
2. When we tried to describe pattern matching for a multiset, we noticed the importance of non-linear patterns, user-customizability of pattern-match algorithms, and polymorphic patterns (Section 2.2, 2.3).
3. When we defined the `comb` function, we got an idea of loop patterns (Section 3.2.6).
4. When we defined the `uniq` function, we got an idea of sequential patterns (Section 3.2.7).

After we noticed the importance of pattern matching with multiple results, we consequently noticed that pattern matching with multiple results is useful for pattern matching non-free data types. Then, we noticed the importance of non-linear patterns, user-customizability of pattern-match algorithms, and polymorphic patterns. When redefining various list functions using the pattern-match facility that satisfies all the above features, we got ideas of loop patterns and sequential patterns.

We have noticed the gaps also 4 times for tensor calculus.

5. When we tried to define tensor multiplication that supports index notation, we found that the definition is not easy (Section 1.3, 5.3).
6. When we tried to write down a formula that contains differential forms, we found it difficult (Section 5.4).
7. When we tried to declare symmetries of a tensor in a program, we noticed that we had no concise notation for this purpose (Section 5.3.7).
8. When we defined the covariant derivative operator, we noticed our description verbose (Section 5.3.8).

We noticed these gaps while we tried to write down formulae in a textbook of differential geometry as many as possible. All these gaps derive from the fact that we cannot directly write down formulae in programs.

Writing many programs for the fields in which many researchers of programming languages still do not have an interest is an easy way to find the cognitive gaps. In mathematics, there are still many calculations that we have never tried to compute using computers.

6.2.2 Step 2: Design New Syntax for Describing Algorithms in a Form Close to Our Recognition

We have designed several new syntax constructs for pattern matching.

1. We designed the `matchAll` expression that collects all the pattern-match results. It enables users to handle multiple pattern-match results (Section 2.3.1).

2. We designed the value pattern to represent non-linear patterns (Section 2.3.2).

3. We designed the pattern-match expressions to take an additional argument matcher. This additional argument is necessary for polymorphic patterns (Section 2.3.3).

4. We designed the `matcher` expression that enables users to customize a pattern-match algorithm (Section 2.5).

5. We designed the loop pattern and sequential pattern (Section 3.2.6, 3.2.7).

Most of the above syntax arose naturally from the reasons for the gaps that we analyzed when we noticed the gaps. The patter-match expressions `matchAll` and `match` are designed by extending the existing pattern-match expressions for algebraic data types. On the other hand, we made much effort to design syntax for defining matchers. This is because the semantics of pattern-match expressions are strongly connected to the syntax for defining matchers.

We have designed several new syntax constructs for tensor calculus.

6. We designed how to describe formulae that uses tensor index notation (Section 5.3.8).

7. We designed syntax for declaring tensor symmetries (Section 5.3.7).

For designing how to describe index notation in programming, we have not made much effort because we can just import notations from mathematics. On the other hand, it took a lot of time to get an idea for declaring tensor symmetries. As mentioned in Section 5.3.7, we got an idea of the syntax from the mathematical notation for commutators $T_{[ij]} = T_{ij} - T_{ji}$.

When we designed the syntax for declaring tensor symmetries, we got the idea that a good notation efficiently embeds algorithm information in a parse tree. The simplest method for declaring tensor symmetries is describing a list of symmetric indices. In contrast, the proposed notation embed this list into a tree by assuming symmetric indices always adjoin and enclosing them by parenthesis. We can get a new good notation by finding a special case that we can convert a list in a parse tree that can be converted to a tree.

In fact, pattern matching is also such an example. In general, functions have multiple arguments and multiple return values. In functional programming, we use a list of the let expressions or do-notation that takes a list of input/output operations to represent sequential applications of functions that have multiple arguments and multiple return values. However, many functions that we deal with take multiple arguments and have a single return value. Thanks to that, we can describe sequential function applications in a tree form like $f(g(x), h(y))$ instead of a list of the let expressions. On the contrary, we can also describe nested applications of the functions that take a single argument and return multiple values in a tree form. Pattern matching is a syntax construct for representing this kind of nested function application. For example, the pattern `$x :: $y :: _`

can be regarded as a nested application of the cons function (::) that has a single argument and two return values. Sweet Egison reverts a pattern in a tree form to a list of input/output operations in a do-expression.

6.2.3 Step 3: Design a Method for Executing New Syntax

We have designed the executing methods for pattern matching as follows.

1. We designed an internal pattern-match algorithm of the proposed pattern-match system as reduction of matching states (Section 2.4).

2. We developed a method for converting the proposed pattern-match expressions to programs that use backtracking monads (Section 4.3).

When we designed the execution methods, we tried to make the semantics of the language as natural as possible. We feel the semantics of a language natural when it is general and simple.

To make the semantics of our language general we designed our pattern-match facility for general non-free data types. We noticed the importance of user-customizable non-linear pattern matching with multiple results when writing a program that deals with lists and multisets. However, we do not limit our pattern-match facility to the collection data types.

To make the semantics of our language simple we chose backtracking as the base algorithm for pattern matching. The backtracking algorithm is the most efficient and simplest for handling non-linear pattern matching with multiple results. Not only that, the backtracking algorithm does not require users complicated descriptions for defining their own patterns.

We have designed the executing methods for tensor calculus as follows.

4. We designed a set of index reduction rules that enables us to describe function application using index notation (Section 5.3).

5. We designed an execution method of programs that declare a tensor using index notation (Section 5.3.6).

6. We designed an execution method of programs that define a tensor operator using index notation (Section 5.3.8).

7. We designed an execution method of programs that declare symmetries of tensors (Section 5.3.7).

8. We designed index completion rules that enables us to describe formulae that contain differential forms (Section 5.4).

Most of the notations for tensor calculus become executable by giving a meaning to undefined programs. In mathematics, symbolic index rules are defined for each tensor operator. We simplified the symbolic index rules so that we can apply them for all the operators.

6.3 Contributions

In this thesis, we have proposed new programming facilities and styles that widen the range of algorithms that we can describe intuitively. To this end, we found algorithms whose description is more complicated than necessary, analyzed the reasons, and designed new programming facilities. Utilizing these new programming facilities, we wrote programs in new programming styles, found algorithms

whose description is still complicated, and extend our new programming facilities. We conducted such research on two independent themes, pattern matching for non-free data types and importing notations from tensor calculus. We explained our work for pattern matching in Chapters 2, 3, and 4, and for tensor calculus in Chapter 5.

6.4 Future Work

This section catalogs and discuss the next goals and research questions we found while developing our proposals.

6.4.1 Increase Application of Pattern-Match-Oriented Programming

Development of new application examples of Egison is important to convince a lot of programmers that Egison is useful for practical use and to make them to think about new application examples. Implementing various algorithms in Egison helps us to clarify what pattern-match-oriented programming is and invent new features of programming languages. We have been implementing SAT algorithms as a representative of algorithms. We explained our implementation of the Davis-Putnam algorithm in Section 3.4.1. We also implemented the CDCL algorithm that has been the basis of the latest SAT algorithm. If we try to implement the state-of-art algorithm, we want to manage memory usage and use our pattern-match facility in system programming languages such as Rust.

Applying our pattern-match facility to query databases is another promising application. As a first step, we are planning to implement software that translates our pattern-match expressions to the existing query languages.

6.4.2 Implement the Proposed Pattern Matching in Other Programming Languages

First, we plan to implement our pattern-match facility as a GHC plugin and GHC extension. We have implemented our pattern-match facility as a Haskell library in Sweet Egison introduced in Chapter 4. However, this library has limitations due to meta-programming capability of Haskell. To begin with, quasi-quoting each match clause is redundant. Not only that, we cannot nest pattern-match expressions because we cannot nest quasi-quotes. We may avoid these problems by implementing our pattern-match facility as a GHC plugin and GHC extension. As a first step, we are surveying to confirm that we can implement our pattern-match facility as a GHC plugin.

Second, we plan to embed Egison pattern matching into Rust, a system programming language with strong meta-programming capability. Embedding our pattern-match facility into a system programming language that allows manual memory management is an interesting research topic. We found several research questions.

- Can we implement more efficient pattern-match algorithms by manually managing memory usage?
- How to design a pattern-match system limiting the capability of our pattern-match facility for avoiding inefficient memory copies?

6.4.3 Proof Writing Language

In the future, mathematical proofs will be written and checked on computers. It is currently not common among mathematics researchers because we cannot yet describe proofs on computers as concisely as in natural languages and formulae. It is known as Curry-Howard correspondence; there is a strong relationship between computer programs and mathematical proofs. The syntax that is useful for a concise description of programs is also useful for a concise description of proofs. Pattern matching is also an important syntactic construct for intuitive descriptions of proofs. A vast effort has been devoted to introducing pattern matching to theorem proving systems [28, 43, 27]. Most modern theorem provers such as Coq [83], Agda [82], and Lean [29] have a pattern-match facility for algebraic data types. Applying the pattern-match facility for non-free data types proposed in this thesis can make the descriptions of proofs more concise. With these ideas in mind, we are designing a proof writing language with our pattern-match facility [36].

6.4.4 Pattern Matching for Wider Range of Data Types

We focused on the data abstraction for non-free data types in this thesis. However, there must be more data types waiting to be abstracted. There are still data types we cannot apply pattern matching. For example, it is difficult to describe intuitive patterns for multi-dimensional data, such as two-dimensional arrays. This is because we cannot embed patterns for these data types into a syntax tree.

6.4.5 Import More Notations from Mathematics

In this thesis, we have imported notations in tensor calculus into programming. However, there are still many notations in mathematics that are useful but not yet introduced into programming. Importing these notations is not only useful but also leads us to deepen our understanding on mathematical notations.

6.4.6 Expressive Complexity

There are no established methods for comparing the readability and writability of multiple programs written in different styles. The lack of such a method is considered one of the reasons why the development of new notations is not very active in programming languages. We always found difficulty in evaluating the features of programming languages proposed in this thesis. We believe we have provided reasonable evidence to show our proposals are useful, but these pieces of evidence are subjective and not mathematically formal. With these ideas in mind, we are considering a method for measuring the readability and writability of a program. We call readability and writability of a program *expressive complexity* inspired by the computational complexity of algorithms. This section introduces the current ideas.

Let A and B be programs we want to compare. We assume that A and B implement the same algorithm and can be compiled to the same machine code. The reason of this assumption is that it is nonsense to compare the expressive complexities of programs that implement different algorithms even if they have the same input and output.

First, we investigated whether we can use the number of tokens, variables, and local variables, or the depth of a parse tree for expressive complexity. However, we

found it difficult because they are a constant for each program and the difference between these constants is unconvincing to measure the readability of programs.

Currently, we are investigating conversions between A and B. In general, the complexity of the conversion from A to B and B to A is not equal. For example, let A be a pattern-match-oriented style program and B be a program that uses backtracking monads. In this case, the conversion A from B is relatively easier than B to A. One reason for this asymmetry is that we can convert all the pattern-match expressions to programs that use backtracking monads, but the opposite is not true. We are investigating how we can utilize this asymmetry for calculating expressive complexity.

References

[1] Attributes::attnf - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/message/Attributes/attnf.html. [Online; accessed 14-June-2018].

[2] Introduction to Patterns - Wolfram Language Documentation. http://reference.wolfram.com/language/tutorial/Introduction-Patterns.html. [Online; accessed 14-June-2018].

[3] Orderless - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Orderless.html. [Online; accessed 14-June-2018].

[4] PAKCS. https://www.informatik.uni-kiel.de/~pakcs/. [Online; accessed 14-June-2018].

[5] ViewPatterns - GHC. https://ghc.haskell.org/trac/ghc/wiki/ViewPatterns, 2015. [Online; accessed 30-Jan-2020].

[6] The Neo4j Manual v2.3.3. https://neo4j.com/docs/stable/index.html, 2016. [Online; accessed 30-Jan-2020].

[7] XML Path Language (XPath) 3.1. https://www.w3.org/TR/2017/REC-xpath-31-20170321, 2017. [Online; accessed 30-Jan-2020].

[8] Listable - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Listable.html, 2020. [Online; accessed 30-Jan-2020].

[9] Maxima - a Computer Algebra System. http://maxima.sourceforge.net/, 2020. [Online; accessed 30-Jan-2020].

[10] Module - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Module.html, 2020. [Online; accessed 30-Jan-2020].

[11] NumPy. http://www.numpy.org/, 2020. [Online; accessed 30-Jan-2020].

[12] Sum - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Sum.html, 2020.

[13] Table - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Table.html, 2020.

[14] Universal functions (ufunc) — NumPy v1.16 Manual. https://docs.scipy.org/doc/numpy/reference/ufuncs.html, 2020. [Online; accessed 30-Jan-2020].

[15] SageMath - Open-Source Mathematical Software System. https://www.sagemath.org/, 2021. Accessed: 11-March-2021.

[16] Krister Åhlander. Einstein summation for multidimensional arrays. *Computers & Mathematics with Applications*, 44(8-9):1007–1017, 2002.

[17] Sergio Antoy. Constructor-Based Conditional Narrowing. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Florence, Italy*, PPDP, pages 199–206. ACM, 2001.

[18] Sergio Antoy. Programming with narrowing: A tutorial. *Journal of Symbolic Computation*, 45(5):501–522, 2010.

[19] John W. Backus. The history of FORTRAN I, II and III. *IEEE Ann. Hist. Comput.*, 1(1):21–37, 1979.

[20] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[21] Robert Bernecky. An introduction to function rank. In Leroy J. Dickey and Lynne C. Shaw, editors, *Proceedings of the international conference on APL, APL 1988, Sydney, Australia, February 1-5, 1988*, pages 39–43. ACM, 1988.

[22] Richard S. Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall International series in computer science. Prentice Hall, 1988.

[23] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. Kics2: A new compiler from curry to haskell. In Herbert Kuchen, editor, *Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*, volume 6816 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[24] Rod M. Burstall. Proving Properties of Programs by Structural Induction. *The Computer Journal*, 12(1):41–48, 1969.

[25] Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE: an experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming, Stanford, California, United States*, LFP, pages 136–143. ACM, 1980.

[26] 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, Ann Arbor, Michigan, United States*, SIGFIDET, pages 249–264. ACM, 1974.

[27] Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017.

[28] Thierry Coquand. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, volume 92, pages 66–79. Citeseer, 1992.

[29] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[30] Satoshi Egi. The Egison Programming Language. `http://www.egison.org/`, 2011. [Online; accessed 14-June-2018].

[31] Satoshi Egi. Egison Mathematics Notebook. `http://www.egison.org/math/`, 2017. [Online; accessed 14-June-2018].

[32] Satoshi Egi. Scalar and Tensor Parameters for Importing Tensor Index Notation including Einstein Summation Notation. In *Proceedings of the 18th Scheme and Functional Programming Workshop, Oxford, United Kingdom, Co-located with ICFP*, 2017.

[33] Satoshi Egi. Loop Patterns: Extension of Kleene Star Operator for More Expressive Pattern Matching against Arbitrary Data Structures. In *Proceedings of the 19th Scheme and Functional Programming Workshop, Saint Louis, Missouri, United States, Co-located with ICFP*, 2018.

[34] Satoshi Egi. Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types. In *Proceedings of the 20th Scheme and Functional Programming Workshop, Berlin, Germany, Co-located with ICFP*, 2019.

[35] Satoshi Egi. egison/backtracking: Backtracking Monad. `https://hackage.haskell.org/package/backtracking`, 2020. [Online; accessed 10-Oct-2021].

[36] Satoshi Egi. Pattern-match-oriented proof writing language. In Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix, editors, *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, pages 223–224. ACM, 2020.

[37] Satoshi Egi, Yoshiaki Maeda, and Steven Rosenberg. Diffeomorphism groups of circle bundles over symplectic manifolds. *arXiv preprint arXiv:2011.01800*, 2020.

[38] Satoshi Egi and Yuichi Nishiwaki. Non-linear pattern matching with backtracking for non-free data types. In *Proceedings of APLAS 2018 - Asian Symposium on Programming Languages and Systems, Wellington, New Zealand*, volume 11275 of *LNCS*, pages 3–23. Springer, 2018.

[39] Martin Erwig. Active patterns. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL 1996), Bad Godesberg, Germany*, volume 1268 of *LNCS*, pages 21–40. Springer, 1996.

[40] Martin Erwig. Functional programming with graphs. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 52–65. ACM, 1997.

[41] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 11–22. ACM, 2009.

[42] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. The ITensor software library for tensor network calculations, 2020.

[43] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

[44] Michael Hanus. Multi-paradigm declarative languages. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Porto, Portugal*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.

[45] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proceedings of ILPS'95 Workshop on Visions for the Future of Logic Programmming, Portland, Oregon, United States*, pages 95–107, 1995.

[46] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[47] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[48] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN History of Programming Languages Conference, San Diego, California, United States*, HOPL, pages 1–55. ACM, 2007.

[49] Thomas J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Civil and Mechanical Engineering. Dover, 2000.

[50] Yuwang Ji, Qiang Wang, Xuan Li, and Jie Liu. A survey on tensor techniques and applications in machine learning. *IEEE Access*, 7:162950–162990, 2019.

[51] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly Types: Type Inference for Generalised Algebraic Data Types. Technical report, Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.

[52] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of lisp. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 231–270. ACM, 1993.

[53] Jerzy Karczmarczuk. Functional coding of differential forms. In *Scottish Workshop on Functional Programming*, 1999.

[54] Akira Kawata. egison/typed-egison: Typed Egison. `https://github.com/egison/typed-egison`, 2018. [Online; accessed 10-Oct-2019].

[55] Gordon L. Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John H. Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Trans. Vis. Comput. Graph.*, 22(1):867–876, 2016.

[56] Mayuko Kori and Satoshi Egi. egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. `https://github.com/egison/egison-haskell`, 2019. [Online; accessed 21-Feb-2019].

[57] Anna V. Korolkova, Dmitry S. Kulyabov, and Leonid A. Sevastyanov. Tensor computations in computer algebra systems. *Program. Comput. Softw.*, 39(3):135–142, 2013.

[58] Manuel Krebber. Non-linear associative-commutative many-to-one pattern matching with sequence variables. *CoRR*, abs/1705.00907, 2017.

[59] John M Lee. Ricci: Mathematica package for doing tensor calculations in differential geometry. `https://sites.math.washington.edu/~lee/Ricci/`, 2016.

[60] Yoshiaki Maeda, Steven Rosenberg, and Fabián Torres-Ardila. Computation of the Wodzicki-Chern-Simons form in local coordinates. Computations for S^1 actions on $S^2 \times S^3$. `http://math.bu.edu/people/sr/articles/ComputationsChernSimonsS2xS3_July_1_2010.pdf`, 2010.

[61] Yoshiaki Maeda, Steven Rosenberg, and Fabián Torres-Ardila. The geometry of loop spaces II: Characteristic classes. *Advances in Mathematics*, 287:485–518, 2016.

[62] John McCarthy. Symbol manipulating language - revisions of the language. *MIT AI Lab. AI Memo No. 4*, 1958.

[63] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978, Los Angeles, California, USA*, pages 173–185. Academic Press / ACM, 1978.

[64] Takayuki Muranushi, Seiya Nishizawa, Hirofumi Tomita, Keigo Nitadori, Masaki Iwasawa, Yutaka Maruyama, Hisashi Yashiro, Yoshifumi Nakamura, Hideyuki Hotta, Junichiro Makino, Natsuki Hosono, and Hikaru Inoue. Automatic generation of efficient codes from mathematical descriptions of stencil computation. In David Duke and Yukiyoshi Kameyama, editors, *Proceedings of the 5th International Workshop on Functional High-Performance Computing, FHPC@ICFP 2016, Nara, Japan, September 22, 2016*, pages 17–22. ACM, 2016.

[65] Chris Okasaki. Views for standard ml. In *SIGPLAN Workshop on ML*, pages 14–23. Citeseer, 1998.

[66] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia, United States*, volume 4273 of *LNCS*, pages 30–43. Springer, 2006.

[67] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[68] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[69] Reiner Pope and Brent Yorgey. 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].

[70] Christian Queinnec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In *Proceedings of Programming Language Implementation and Logic Programming (PLIP 1990), Linköping, Sweden*, volume 456 of *LNCS*, pages 340–357. Springer, 1990.

[71] MMG Ricci and Tullio Levi-Civita. Méthodes de calcul différentiel absolu et leurs applications. *Mathematische Annalen*, 54(1-2):125–201, 1900.

[72] Marko A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, Pennsylvania, United States*, DBLP, pages 1–10. ACM, 2015.

[73] Min Ru. Hodge theory on Riemannian manifolds. `https://www.math.uh.edu/~minru/Riemann08/hodgetheory.pdf`, 2008. [Online; accessed 30-Jan-2020].

[74] Bernard F Schutz. *Geometrical methods of mathematical physics*. Cambridge university press, 1980.

[75] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[76] Olin Shivers. SRFI 1: List Library. `https://srfi.schemers.org/srfi-1/srfi-1.html`, 1999. [Online; accessed 30-Jan-2020].

[77] Edgar Solomonik and Torsten Hoefler. Sparse tensor algebra as a parallel programming model. *CoRR*, abs/1512.00066, 2015.

[78] J. Michael Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19(3-4):469–487, 2009.

[79] Gerald Jay Sussman and Jack Wisdom. *Functional differential geometry*. MIT Press, 2013.

[80] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 29–40. ACM, 2007.

[81] GHC team. 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].

[82] The Agda Development Team. Agda, 2020.

[83] The Coq Development Team. Coq, 2020.

[84] Simon J. Thompson. Laws in miranda. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, Massachusetts, United States*, LFP, pages 1–12. ACM, 1986.

[85] Simon J. Thompson. Lawful Functions and Program Verification in Miranda. *Science of Computer Programming*, 13:181–218, 1989.

[86] Sam Tobin-Hochstadt. Extensible Pattern Matching in an Extensible Language. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages, Alphen aan den Rijn, The Netherlands*, IFL, 2010.

[87] Viktor T. Toth. Tensor manipulation in GPL maxima. *CoRR*, abs/cs/0503073, 2005.

[88] Phil Trinder and Philip Wadler. Improving List Comprehension Database Queries. In *Proceedings of the 4th IEEE Region 10 International Conference TENCON*, pages 186–192. IEEE, IEEE, 1989.

[89] Mark Tullsen. First class patterns. In *Proceedings of the 2nd International Symposium on Practical Aspects of Declarative Languages (PADL 2000), Boston, Massachusetts, United States*, volume 1753 of *LNCS*, pages 1–15. Springer, 2000.

[90] David A. Turner. Miranda: A Non-Strict Functional language with Polymorphic Types. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA 1985), Nancy, France*, volume 201 of *LNCS*, pages 1–16. Springer, 1985.

[91] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[92] 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, Munich, Germany*, POPL, pages 307–313, 1987.