日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

正規形を持たないデータ型に対するパターンマッチング

江木 聡志

データ型には，1 つの定まった正規形を持たないデータ型がある．例えば，集合や多重集合などがそうであり，{a,b,c} という集合は，{b,a,c} や {c,b,a}，{a,a,b,c} という集合と同値である．このような正規形を持たないデータ型に対してパターンマッチを行うには，パターンマッチの際にバックトラック (探索) を行う必要がある．バックトラックを行うパターンマッチングは理論的には既に知られているが，その有用性は今まであまり示されていなかった．本研究は，パターンマッチの際にパターンの左側で現れたパターン変数に束縛された値を，それより右側のパターンで参照できる仕組みや，パターンマッチの際に不必要な探索をしないようにバックトラックを制御する仕組みを提案し，バックトラックを用いるパターンマッチングを実用的なものにし，その有用性を示した．本研究で提案されたアイデアは全てプログラミング言語 Egison に組み込まれている．

1 はじめに

様々なデータ型を自在に扱えることは，アルゴリズムを直感的に表現するために重要なことである．あるデータ型を自在に扱えるということは，言い換えると，そのデータ型のデータのコンストラクト，デコンストラクトが自在に行えるということである．よって，データの直感的なコンストラクト，デコンストラクトの表現を考えるのは重要なことである．

本研究は，パターンマッチを行う式の表現を中心にこの問題を考える．パターンマッチはデータのデコンストラクタと値の束縛を同時に行う非常に直感的なデコンストラクト操作であるからである．

本章では，まず現状のパターンマッチの問題点を提示し，その後，本研究によるその改善の方法を Egison によるサンプルコードとともに示す．

1.1 現状の問題点

パターンマッチは，ほとんどの関数型プログラミング言語に組み込まれているが，これらのパターンマッチには大きな弱点がある．それは 1 つの定まった正規形を持たないデータ型に対するパターンマッチが直接的に行えないことである．

一般的な関数型プログラミング言語では，新しいデータ型を定義する際，すでに定義されているデータ型の直積や直和を用いて再帰的に定義される．そして，その定義に基づきそのデータ型をデコンストラクトする．すべてのデータ型がこのように直積と直和と再帰で定義できるのであれば，これはもっともエレガントなアプローチである．しかし，正規形を持たないデータ型は，そのように単純には定義することができない．

リストや集合，多重集合のような任意個の同じ種類の要素からなるデータはコレクションデータと呼ばれている．コレクションデータ型は，多くのアルゴリズムに現れるが，一般的に正規形を持たない．リストであれば，よく知られている `Nil` と `Cons` を使った再帰的定義が可能であるが，集合や多重集合などは同じようにしてもうまくいかない．例えば，{a,b,c} という集合に対して {b,a,c} や {c,b,a}，{a,a,b,c} とい

Pattern Matching for Unfree Data Types

Satoshi Egi, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, Tokyo University.

う集合も同値であり，一つの決まった形を持たない．同じ要素の重複の扱いや，要素の順序関係の扱いは型の直積，直和，再帰だけでは表現しきれない．

このような正規形を持たないデータに対するデコンストラクトを行うには，バックトラッキングを行う必要がある．そのため単純な表現で強力なパターンマッチを実現するのは難しい．この論文では，このような正規形を持たないデータ型に対しても強力なパターンマッチの表現を提示する．

1.2 例

以下のコードは，Egison でパターンマッチを行うコードである．

```
(match {2 7 7 2 7} (Multiset Int)
  {[<cons $m
     <cons ,m
      <cons ,m
       !<cons $n
         <cons ,n
          <nil>>>>>>
    <ok>]
   [ _ <ko>]})

=> <ok>
```

この式は，{2 7 7 2 7} というコレクションを，`(Multiset Int)` としてパターンマッチするという意味である．パターンマッチのターゲット (*target*) である {2 7 7 2 7} が要素 5 つからなるコレクションであり，同じ数を 3 つ含み，残り 2 つの数も同じ数である場合，1 つ目のマッチ節のパターンにマッチし，値`<ok>`を返し，そうでない場合，2 つ目のパターンにマッチし，値`<ko>`を返す．2 つ目のマッチ節のパターンは，ターゲットが何でもパターンマッチに成功する．'_' はワイルドカード (*wild card*) であり，ターゲットが何でもパターンマッチ成功する．

上の例をみてわかるとおり，Egison には 4 種類の括弧がある．

'(', ')' で囲まれた式は，言語に組み込まれている構文を適用する式を表すのと，ユーザ定義した関数を適用する式を表すのに使われる．囲まれた式のうちで先頭にあるものがオペレータで，それ以降の式が引数となる．

'<', '>' で囲まれた式は，データコンストラクタの適用を表す．囲まれた式のうちで先頭の式がデータコンストラクタで，それ以降の式が引数となる．'<', '>' で囲まれた式は，パターンコンストラクタ (*pattern constructor*) の適用を表すのにも使われる．

'[', ']' で囲まれた式はタプルを表す．Egison のタプルは多値として使うことができる．1 つの式しか含んでいないタプルはその 1 つの式と同じものとして扱われる．

```
1 = [1] = [[1]] = ...
```

'{', '}' で囲まれた式はコレクション (任意個の同じ種類の要素からなるデータ) を表す．'{', '}' で囲まれている式には先頭に'@' がついていない式とついている式との 2 種類がある．'@' がついていない式は，全体のコレクションの要素の 1 つとして扱われる．'@' がついている式は，全体のコレクションの部分コレクション (subcollection) として扱われる．

```
{1 @{2 3 4} @{5 @{6} 7} 8} = {1 2 3 4 5 6 7 8}
```

`match` 構文は 1 つ目の引数にパターンマッチのターゲットを，2 つ目の引数にパターンマッチを行う際の型 (*type*) を，3 つめの引数にマッチ節 (*match clause*) のコレクションをとる．`(Multiset Int)` という式は，整数の多重集合という型を表している．多重集合とは，要素の順序関係は無視するが，重複は考えるコレクションデータ型のことである．2 つ目の引数にパターンマッチを行う際の型を取るところが，既存のパターンマッチの表現との大きな違いである．

1 つ目のマッチ節のパターンは，ターゲットが要素 5 つのコレクションで同じ要素が 3 つあり，残り 2 つの要素も同じである場合にパターンマッチに成功するパターンである．Egison ではパターンの左側から順にパターンマッチしていく．Egison のパターンマッチは non-left-linear になっていて，パターンに現れたパターン変数に束縛される値を，それより右側で参照できる．'$' が先頭についたリテラルは，パターン変数 (*pattern variable*) である．パターン変数からなるパターンを変数パターン (*variable pattern*) と呼ぶ．','

が先頭についたパターンは，バリューパターン (*value pattern*) であり，評価されその値とターゲットを比較して同じだった場合，パターンマッチに成功する．その際の評価で，左側のパターン変数に束縛された値を参照することができる．'!' が先頭についたパターンは，カットパターン (*cut pattern*) である．カットパターンは不必要なバックトラックをなくすために使われる．この場合は，5 つの数のうちに同じ数が 3 つあるなら，その数は 1 通りしかないので，もしその組み合わせが見つかれば，それ以上のバックトラックは必要ないことを表現している．

カットパターンを使わずに書くと，先ほどのコードは以下のコードと同時である．

```
(match {2 7 7 2 7} (Multiset Int)
  {[<cons $m
     <cons ,m
      <cons ,m
       $tmp1>>>
    (match tmp1 (Multiset Int)
      {[<cons $n
         <cons ,n
          <nil>>>
        <ok>]})]
   [_ <ko>]})
```

Egison では，パターンはファーストクラスオブジェクトであり，以下のように，パターンも他の式と同じように評価したり，関数の引数として渡したりすることができる．

```
(let {[$pat <cons ,1 <nil>>]}
  (match {1} (Multiset Int)
    {[pat <ok>]
     [_ <ko>]}))

=> <ok>

(let {[$loop <cons ,1 (of {<nil> loop})>]}
  (match {1 1 1 1} (Multiset Int)
    {[loop <ok>]
     [_ <ko>]}))

=> <ok>
```

2 つ目の例の `let` 式に現れる `of` 式によって定義されるパターンは `of` パターン (*of pattern*) と呼ばれるパターンである．引数のコレクションに含まれるパターンのどれかにパターンマッチすればパターンマッチに成功する．この例では，`loop` は `cons` で 1 を 1 つ以上任意個つないだパターンとなる．Egison の `let` 式は相互再帰的な束縛を行うため，このように再帰的な定義を行える．

Egison によるパターンマッチでは，結果として可能な束縛が複数ある場合がある．`match-all` 式を使うと，複数の束縛結果それぞれについて，処理を実行することができる．`match-all` 式は，`match` 式と同じく 1 つ目の引数にターゲットを，2 つ目の引数に型をとる．ただ違うのは，3 つめの引数で，マッチ節のコレクションではなく，単独のマッチ節をとる．パターンとターゲットを，指定された型としてパターンマッチを実行し，得られた可能な束縛のコレクションの全ての束縛について，それぞれマッチ節の式を実行し，その結果をコレクションにして返す．

以下は，`match-all` 式を用いて，コレクションデータを色々なコレクション型でパターンマッチしている例である．パターンコンストラクタ `join` は引数を 2 つ取り，1 つ目の引数にマッチした値と 2 つ目の引数にマッチした値を合わせた要素が，ターゲットと同値であればパターンマッチに成功する．

```
(match-all {1 2 3} (List Int)
  [<join $hs $ts> [hs ts]]))

=> {[{} {1 2 3}] [{1} {2 3}]
    [{1 2} {3}] [{1 2 3} {}]}

(match-all {1 2 3} (Multiset Int)
  [<join $hs $ts> [hs ts]]))

=> {[{} {1 2 3}] [{3} {1 2}] [{2} {1 3}]
    [{2 3} {1}] [{1} {2 3}] [{1 3} {2}]
    [{1 2} {3}] [{1 2 3} {}]}

(match-all {1 2 3} (Set Int)
  [<join $hs $ts> [hs ts]]))

=> {[{} {1 2 3}] [{3} {1 2}] [{3} {1 2 3}]
    [{2} {1 3}] [{2} {1 3 2}] [{2 3} {1}]
    [{2 3} {1 3}] [{2 3} {1 2}] [{2 3} {1 2 3}]
    [{1} {2 3}] [{1} {2 3 1}] [{1 3} {2}]
    [{1 3} {2 3}] [{1 3} {2 1}] [{1 3} {2 1 3}]
    [{1 2} {3}] [{1 2} {3 2}] [{1 2} {3 1}]
    [{1 2} {3 1 2}] [{1 2 3} {}] [{1 2 3} {3}]
    [{1 2 3} {2}] [{1 2 3} {2 3}] [{1 2 3} {1}]
```

```
[{1 2 3} {1 3}] [{1 2 3} {1 2}]
[{1 2 3} {1 2 3}]}
```

1.3 本論文の構成

本論文の 2 章以降は以下のような構成になっている．2 章では，本章で示したパターンマッチの式がどのような仕組みで動くのか，その仕組みを説明する．3 章では，Egison のパターンマッチにより表現が簡潔になる具体的な例をあげる．4 章では，関連研究との比較をする．5 章では，本研究の貢献，これからの展望についてまとめる．

2 パターンマッチの仕組み

前章で説明したパターンマッチの式が動くのは，型ごとにどのようにパターンマッチを行うのか記述しているからである．本章では，型の定義の方法を示しながら，実際にどのような流れでパターンマッチが行われるのかを説明する．

多重集合 (同じ要素の重複を考慮する集合) の型の定義を理解すれば，型定義の際に必要なことが全て理解できる．したがって，ここでは多重集合の型の定義の例を用いて説明する．図 1 は，Egison で多重集合の型を定義したものである．

`remove` は型を引数に取り，1 つ目の引数の値の要素を 2 つ目の引数のコレクションから取り除く関数を返す．`remove-collection` は型を引数に取り，1 つ目の引数のコレクションの要素を 2 つ目の引数のコレクションから取り除く関数を返す．

`Multiset` は，型を 1 つ受け取って，その型の多重集合の型を返す関数となっている．例えば，`(Multiset Int)` は数の多重集合という型になり，`(Multiset (Multiset Int))` は数の多重集合の多重集合という型になる．

型は `type` 式を用いて定義される．`type` 式は引数に `let` 式の 1 つ目の引数と同じく，パターン変数と式のタプルのコレクションをとる．`let` 式と同じく相互再帰的な値の定義を行うことができる．`type-ref` 式を用いると `type` 式で定義した値を参照することができる．

`Multiset` の型の定義では，`var-match` 関数と，`inductive-match` 関数，`equal?`関数が定義されている．この 3 つの関数を使って，マッチ関数 (*match function*) が構成される．マッチ関数とは，パターンとターゲットを受け取って，可能な束縛フレームのコレクションを返す関数である．`var-match` 関数と，`inductive-match` 関数，`equal?`関数の定義から，マッチ関数が自動で構成される．

`var-match` 関数，`inductive-match` 関数，`equal?`関数についてそれぞれ説明する．

`var-match` 関数は，パターンが変数パターンである場合に使われる．`var-match` 関数にターゲットの値を適用して得られる返り値のコレクションの要素が，そのパターン変数の取りうる値となる．この `Multiset` の定義の例だと，パターン変数の取りうる値は，ターゲットの値だけということが記述されている．

`inductive-match` 関数は，パターンがインダクティブパターン (*inductive pattern*) である場合に使われる．インダクティブパターンとは，パターンコンストラクタを用いて構成されたパターンのことである．`inductive-match` 関数についての説明は長くなるので，`equal?`関数を説明したすぐ後でまた説明する．

`equal?`関数は，パターンがバリューパターンである場合に使われる．`equal?`関数に，バリューパターンの中身の値とターゲットとを適用した返り値の真偽値が，パターンマッチ可能かどうかを示す．`Multiset` の定義の場合だと，順序関係なく同じ要素を含んでいたら`<true>`を返し，そうでなかったら`<false` を返すように定義されている．この定義では，`equal?`関数の中で再帰的に `Multiset` が呼び出されている．

では，再び `inductive-match` 関数について説明する．`inductive-match` 関数は，`deconstructor` 式を用いて定義される．`deconstructor` 式は，デコンストラクトマッチ節 (*deconstructor-match clause*) のコレクションを引数に取る．デコンストラクトマッチ節は，パターンコンストラクタ，評価したら型のタプルを返す式，プリミティブマッチ節 (*primitive match clause*) のコレクションからなる．例として，まず `Multiset` の定義の `cons` のデコンストラクトマッチ節をみる．デコンストラクトマッチ節の 2 つ目の要

```
(define $Multiset
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (deconstructor
          {[nil []
            {[{} {[]}]
             [_ {}]}]
           [cons [a (Multiset a)]
            {[$tgt (map (lambda [$t] [t ((remove a) tgt t)]) tgt)]}]
           [join [(Multiset a) (Multiset a)]
            {[$tgt (map (lambda [$ts] [ts ((remove-collection a) tgt ts)])
                        (subcollections tgt))]}]})]
       [$equal?
        (lambda [$val $tgt]
          (match [val tgt] [(Multiset a) (Multiset a)]
            {[[<nil> <nil>] <true>]
             [[<cons $x $xs> <cons ,x ,xs>] <true>]
             [[_ _] <false>]}))]})))
```

図 1　Multiset の型の定義

素に `[a (Multiset a)]` とあるのは，パターンのパターンコンストラクタが `cons` であった場合，1 つ目の引数は型 `a` として，2 つ目の引数は型 `(Multiset a)` として，再帰的にパターンマッチするということを表している．

プリミティブマッチ節は，プリミティブパターン (*primitive pattern*) と式のタプルとして表される．マッチ式のターゲットが，プリミティブパターンにマッチしたら，そのプリミティブマッチ節の式を評価した値が，再帰的に行われる次のパターンマッチのターゲットになる．

例の場合だと，型が `(Multiset Int)` で，パターンが`<cons $x $xs>`という形のインダクティブパターンで，ターゲットが `{1 2 3}` というコレクションだった場合，`inductive-match` 関数を使って，パターンマッチの処理が行われる．その際，`inductive-match` 関数の `cons` のデコンストラクトマッチ節にマッチし，さらに，その中の唯一のプリミティブマッチ節にマッチする．このプリミティブマッチ節の式を評価すると，`{[1 {2 3}] [2 {1 3}] [3 {1 2}]}` がその結果として返ってくる．これのそれぞれ 1 つ目の要素を `Int` として，2 つ目の要素を `(Multiset Int)` として，再帰的にパターンマッチを行われる．

プリミティブパターンマッチでは，パターンマッチに成功する場合，可能な束縛結果が一意的に決まるようなパターンマッチしか行えない．プリミティブパターンマッチでは，ひとがプリミティブに行えるデコンストラクトを行えるようになっている．

現状の Egison で，プリミティブパターンこれから説明する 7 つの要素から構成される．

$$
\begin{aligned}
\mathit{primpat} ::= {} & \_ \\
& \mid \mathit{patvar} \\
& \mid \texttt{<}\,\mathit{cons}\ \mathit{primpat}\ \ldots\,\texttt{>} \\
& \mid \texttt{[}\,\mathit{primpat}\ \ldots\,\texttt{]} \\
& \mid \{\} \\
& \mid \{\,\mathit{primpat}\ .\mathit{primpat}\,\} \\
& \mid \{\ .\mathit{primpat}\ \mathit{primpat}\,\}
\end{aligned}
$$

'_' は，ワイルドカードである．ターゲットが何でもパターンマッチ成功する．

pat-var は，パターン変数である．ターゲットが何でもパターンマッチに成功し，値をその変数に束縛します．

`<`*cons prime-pat ...* `>`は，ターゲットが `cons` で指定されたデータコンストラクタで構成されたインダクティブデータである場合にパターンマッチを行う．データコンストラクタの引数について再帰的にパター

ンマッチを行う．

[*prime-pat ...*] は，ターゲットがタプルの場合，パターンマッチを行う．タプルの中身の要素について再帰的にパターンマッチを行う．

{}は，エンプティパターン (*empty pattern*) と呼ばれる．ターゲットが空のコレクションの場合，パターンマッチする．

{*prime-pat .prime-pat*}は，コンスパターン (*cons pattern*) と呼ばれる．ターゲットが 1 つ以上の要素を含むコレクションである場合にリストとしてパターンマッチを行い，1 つ目の *prime-pat* とターゲットの先頭の要素を，2 つ目の *prime-pat* とターゲットの残りの要素コレクションをパターンマッチする．

{*.prime-pat prime-pat*}は，スノックパターン (*snoc pattern*) と呼ばれる．ターゲットが 1 つ以上の要素を含むコレクションである場合にリストとしてパターンマッチを行い，2 つ目の *prime-pat* とターゲットの一番後ろの要素を，1 つ目の *prime-pat* とターゲットの残りの要素コレクションをパターンマッチする．

多重集合の定義と同じように，リストや集合の型の定義もできる．図 2 は，リストの型を定義したものである．`nil` のプリミティブパターンマッチではエンプティパターンが，`cons` のプリミティブパターンマッチではコンスパターンが，`snoc` のプリミティブパターンマッチではスノックパターンが使われている．

3 応用例

この章では，Egison のパターンマッチを利用するとアルゴリズムの表現が簡単になる例をあげる．

3.1 ポーカーの役判定

ポーカーや麻雀の役判定は，人間にとっては非常に簡単である．しかし，多くの言語では，プログラムで表現するのは非常に複雑である．それは，多くの言語では，コレクションをデコンストラクトする際，リストとしてパターンマッチすることしかできないからである．多重集合や集合などパターンマッチをする場合は，個々のパターンについてそれぞれ個別にどのようにデコンストラクトするかを記述しなければならない．Egison では，どのパターンについても実行できるパターンマッチの方法を定義することができるので，非常にシンプルにこれらの操作を表現することができる．

図 3 は，トランプのカードの型の定義を Egison で記述したものである．スートと 13 の剰余環の型を先に定義し，その 2 つを組み合わせてカードの型を定義している．

図 4 は，ポーカーの役判定を行うコードを Egison で記述したものである．カードの多重集合として，カードの組み合わせに対してパターンマッチを行っている．この式は簡潔でありながら，その動きは人間が頭の中で行うパターンマッチと同じである．

3.2 グラフのパターンマッチ

Egison のパターンマッチにより簡潔に表現することができるようになるのは，ポーカーや麻雀のパターンマッチだけではない．例えば，グラフのパターンマッチも直感的に表現することができる．

以下が，グラフの型を Egison で定義したものである．

```
(define $Node Integer)

(define $Graph
  (Multiset [(Multiset [Node Integer])
             Node
             (Multiset [Node Integer])]))
```

グラフはノードについての情報の多重集合として定義される．ノードについての情報は，そのノードを終点とするノードの多重集合とそのノード自身，そのノードを始点とするノードの多重集合のタプルとして表される．このグラフの表現は人間の頭の中のグラフの捉え方と全く同じである．

4 関連研究との比較

データのデコンストラクト，パターンマッチの表現についての研究は多くある．

そのなかでもっとも有名な研究に Views [8] がある．複数の捉え方が可能な抽象データに対して，複数の

```
(define $List
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (deconstructor
          {[nil []
            {[{} {[]}]
             [_ {}]}]
           [cons [a (List a)]
            {[{$x .$xs} {[x xs]}]
             [_ {}]}]
           [snoc [a (List a)]
            {[{.$xs $x} {[x xs]}]
             [_ {}]}]
           [join [(List a) (List a)]
            {[$tgt (let {[$loop (lambda [$ts]
                                  (match ts (List a)
                                    {[<nil> {[{} {}]}]
                                     [<cons $x $xs> {[{} ts] @(map (lambda [$as $bs] [{x @as} bs])
                                                                   (loop xs))}]}))]}
                   (loop tgt))]}]
           [nioj [(List a) (List a)]
            {[$tgt (let {[$loop (lambda [$ts]
                                  (match ts (List a)
                                    {[<nil> {[{} {}]}]
                                     [<snoc $x $xs> {[{} ts] @(map (lambda [$as $bs] [{@as x} bs])
                                                                   (loop xs))}]}))]}
                   (loop tgt))]}]})]
       [$equal? (lambda [$val $tgt]
                  (match [val tgt] [(List a) (List a)]
                    {[[<nil> <nil>] <true>]
                     [[<cons $x $xs> <cons ,x ,xs>]
                      <true>]
                     [[_ _] <false>]}))]})))
```

図 2　List の型の定義

方法でパターンマッチできるようにしたものである．例えば，複素数は実部と虚部の直積と捉えることもできるし，極形式として捉えることもできる．Views は，ユーザーがその二つの間の変換方法を定義しておくと，パターンマッチの際にその変換を行い，もしパターンとターゲットが違う形式でもパターンマッチを行うことができるというものである．論文の中で，例として直交座標形式でも極座標形式でも構成でき，デコンストラクトできる複素数型を定義したり，Cons によるリストとも，Join によるリストともみなせるリスト型を定義したりしている．しかし，Views によるパターンマッチングはバックトラックを行うことができないし，パターンがファーストクラスオブジェクトでもない．そのため，ポーカーの役判定のような複雑なパターンを記述することが難しい．

また，目的が同じ研究に Active Patterns [4] [6] がある．この研究もコンストラクタ毎にデコンストラクトの際のアルゴリズムが記述できる仕組みを考えている．この論文には，多重集合をパターンマッチする例がある．また，Active Patterns を利用してグラフのパターンマッチを行う応用もある [5]．しかし，Active Patterns でもパターンマッチングの際，バックトラックを行うことができない．そのため，Active Patterns には，パターンの先頭から順番に値を確定

```
(define $Suit
  (type
    {[$var-match (lambda [$tgt] {tgt})]
     [$inductive-match
      (deconstructor
        {[spade []
          {[<spade> {[]}]
           [_ {}]}]
         [heart []
          {[<heart> {[]}]
           [_ {}]}]
         [club []
          {[<club> {[]}]
           [_ {}]}]
         [diamond []
          {[<diamond> {[]}]
           [_ {}]}]})]
     [$equal? (lambda [$val $tgt]
                (match [val tgt] [Suit Suit]
                  {[[<spade> <spade>] <true>]
                   [[<heart> <heart>] <true>]
                   [[<club> <club>] <true>]
                   [[<diamond> <diamond>] <true>]
                   [[_ _] <false>]}))]}))

(define $Mod
  (lambda [$m]
    (type
      {[$var-match (lambda [$tgt] {(mod tgt m)})]
       [$equal? (lambda [$val $tgt] (= (mod val m) (mod tgt m)))]})))

(define $Card
  (type
    {[$var-match (lambda [$tgt] {tgt})]
     [$inductive-match
      (deconstructor
        {[card [Suit (Mod 13)]
          {[<card $s $n> {[s n]}]}]})]
     [$equal? (lambda [$val $tgt]
                (match [val tgt] [Card Card]
                  {[[<card $s $n> <card ,s ,n>] <true>]
                   [[_ _] <false>]}))]}))
```

図 3　カードの型の定義

していくパターンしかパターンマッチすることができない．そのため，Active Patterns でもポーカーの役判定のような複雑なパターンを記述することが難しい．

First Class Patterns[7] も，Views を拡張した研究である．名前の通り，パターンがファーストクラスオブジェクトとして扱うようになっている．この論文は，正規形を持たないデータ型に対してのパターンマッチの表現について具体的に言及していないが，いくつかの成功結果があるパターンマッチの提案もしていて本研究のアイデアとかなり近く，本研究はこの研究を発展させたものといえる．First Class Patterns では，パターンコンストラクタにターゲットをデコンストラクトする方法の情報を付与して，その情報をも

```
(define $poker-hands
  (lambda [$Cs]
    (match Cs (Multiset Card)
      {[<cons <card $S $n>
         <cons <card ,S ,(- n 1)>
          <cons <card ,S ,(- n 2)>
           <cons <card ,S ,(- n 3)>
            <cons <card ,S ,(- n 4)>
             !<nil>>>>>>
        <straight-flush>]
       [<cons <card _ $n> <cons <card _ ,n> !<cons <card _ ,n> !<cons <card _ ,n>
         !<cons _ !<nil>>>>>>
        <four-of-kind>]
       [<cons <card _ $m> <cons <card _ ,m> <cons <card _ ,m>
         !<cons <card _ $n> !<cons <card _ ,n> !<nil>>>>>>
        <full-house>]
       [<cons <card $S _>
         !<cons <card ,S _>
           !<cons <card ,S _>
             !<cons <card ,S _>
               !<cons <card ,S _>
                 !<nil>>>>>>
        <flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             !<nil>>>>>>
        <straight>]
       [<cons <card _ $n> <cons <card _ ,n> <cons <card _ ,n>
          <cons _ <cons _ !<nil>>>>>>
        <three-of-kind>]
       [<cons <card _ $m> <cons <card _ ,m>
         !<cons <card _ $n> <cons <card _ ,n>
           !<cons _ !<nil>>>>>>
        <two-pair>]
       [<cons <card _ $n> <cons <card _ ,n>
         <cons _ <cons _ <cons _ !<nil>>>>>>
        <one-pair>]
       [<cons _ <cons _ <cons _ <cons _  <cons _ !<nil>>>>>> <nothing>]}))))
```

図 4　ポーカーの役判定

とにパターンマッチングを行う．First Class Pattern は Haskell の拡張として設計されている．Egison のデコンストラクトマッチ節の内容と同じように，パターンコンストラクタにデコンストラクトの方法を記述し，それぞれのデータ型毎のパターンマッチを実現する．Haskell 組み込みのパターンマッチが，Egison のプリミティブパターンのパターンマッチと対応している．First Class Pattern と比べての Egison の優位性については次章で詳しく述べる．

また，全く別のアプローチに論理型プログラミングのユニフィーケーション [3] [2] を応用したパターンマッチングもある．関数論理型プログラミング (*Functional Logic Programming*) [1] では，論理型プログラミングのユニフィケーションの仕組みをデータのデコンストラクトに使う．論理型プログラミングのユニフィケーションは，本研究の同じくバックトラックを行うため，正規形を持たないデータ型もパターンマッチを行うことができる．しかし，ユニフィケーション

を応用したパターンマッチングは他のパターンマッチの拡張に比べてあまり直感的ではない．例えば，論理関数型プログラミング言語 Curry [1] を紹介する論文には，フォーカード (four of kind) の役判定を行うコードの例があげられている．以下がそのコードである．

```
four (x++[y]++z) | map rank (x++z) =:= [r,r,r,r]
                 = r
                 where r free
```

この four という関数はカードのリストを引数にとり，役が 4 カードだった場合，4 枚あったカードの番号を返す．役が 4 カードでなかった場合は，失敗する．これは，先頭の要素いくつかからなるリスト，それに続く真ん中の要素 1 つからなるリスト，残りの要素からなるリストの 3 つに引数のリストを分解して，引数のリストからどれか要素を 1 つ除いたリストの要素が全て同じ数で，要素の数が 4 個であったらパターンマッチに成功するというようにしている．多重集合や集合などそれぞれのデータ型についてパターンマッチの方法を定義するようになっていなくて，リストとしてパターンマッチを行うので，自動でバックトラッキングしてくれるもののリストとして扱うように翻訳する作業が必要である．

パターンマッチは行うには，パターンとデータを受け取りパターンに現れるパターン変数の取りうる値を計算するの方法を記述すればよい．それに対してユニフィケーションを行うには，パターンを 2 つ受け取り双方向的なパターンマッチを行うためのルールを記述する．アルゴリズムを直感的に表現するという目的には，直接データのデコンストラクトの方法を記述するパターンマッチが自然であるし，また，双方向的なパターンマッチがアルゴリズムの記述に必要になることはないのでパターンマッチで十分である．

5 まとめと今後の課題

5.1 まとめ

Egison による貢献をまとめる．Egison による貢献は大きく以下の 3 つである．これら 3 点については First Class Patterns では実現されていない．

1 つ目の貢献は，パターンの左側のパターン変数に束縛された値をそれより右側で参照できる仕組みやカットパターンのようなパターンマッチの際に不必要な探索をしないようにバックトラックを制御する仕組みをパターンマッチの処理に組み込んで，正規形を持たないデータに対して簡潔なパターンマッチの表現を実現したことである．

2 つ目の貢献は，デコンストラクトの方法の情報をもった型タイプを導入しファーストクラスオブジェクトとして言語に組み込んだことである．これにより，パターンを変えずに，指定する型を変えるだけでパターンマッチを表現することができる．同じプログラム内で，同じコレクションデータをリストとして扱ったり，多重集合として扱ったり，集合として扱ったりすることや，同じ数データを整数として捉えたり，何かの数の剰余環として扱いたくなることは多い．そのような場合に同じパターンコンストラクタを使い回すことができるのは，コードの生産性，可読性を上げる．また，それだけでなく，入れ子構造の型のパターンマッチが直感的が表現できる．例えば，整数の集合の集合という型は，`(Set (Set Int))` というように単純に表現することができ，他の型と同様にパターンマッチも直感的に行える．

3 つ目の貢献は，コレクションデータについて専用のコンストラクタを用意し，デコンストラクトの方法だけでなくコンストラクトの方法についても新たな提案をしたことである．Egison では，コレクションデータに対して新たにコンストラクタと専用のプリミティブパターンを用意した．これにより，特にリストや，多重集合，集合といったコレクションデータに対してのパターンマッチは特に直感的に記述できるようになった．現状の Egison では，コレクションデータに対してしか新たにコンストラクタと専用のプリミティブパターンを用意していないが，ハッシュなどのデータ構造についても同じ考え方で専用のプリミティブパターンを用意すれば，将棋盤やオセロ盤などの直感的なパターンマッチも行えるようになりそうである．

5.2 今後の課題

前節でまとめたいくつかのアイデアにより，正規形を持たないデータ型に対してのパターンマッチがかなり直感的に表現できるようになった．

しかし，前節で述べたとおり，それだけではカバーできていないデータ型もある．例えば，将棋盤やオセロ盤のパターンマッチは現状の Egison では，人間の頭の中のイメージそのままで表現できない．前節で述べたとおり，コレクションデータ以外の，ハッシュなどのデータ構造についても同じ考え方で専用のプリミティブパターンを用意すれば，将棋盤やオセロ盤などの直感的なパターンマッチも行えるようになりそうである．

Egison は動的型付け言語として，現在のところ実装されている．アルゴリズムの表現が直感的になった分，その性質の表現も直感的にできるはずである．Egison の上に，より直感に近い形の型システムの構築を考えたいとも考えている．

謝辞 本論文の内容，文章について多くのアドバイスを頂いた萩谷昌己先生，角谷良彦先生に感謝します．Egison 実装の際に多くのアドバイスをくれ，実装の時間を短縮していただいた平井洋一先輩に感謝します．2 人目の Egisonist となり，Egison のバグを発見したり，コーディングについての新たな知見を教えてくれた本多健太郎君に感謝します．

参 考 文 献

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

[2] A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic (TOCL)*, 9(3):15, 2008.

[3] A. Dovier, A. Policriti, and G. Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2-3):201–234, 1998.

[4] M. Erwig. Active patterns. *Implementation of Functional Languages*, pages 21–40, 1996.

[5] M. Erwig. Functional programming with graphs. In *ACM SIGPLAN Notices*, volume 32, pages 52–65. ACM, 1997.

[6] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, page 40. ACM, 2007.

[7] M. Tullsen. First Class Patterns. *Practical Aspects of Declarative Languages*, pages 1–15, 2000.

[8] P. 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*, page 313. ACM, 1987.