プログラミング言語Egison

江木 聡志
東京大学

Egison は強力なパターンマッチ機能をもつ純粋関数型言語です．Egison を使うと，純粋に帰納的には表せないデータ，例えば，集合や，多重集合，また環や群といった代数構造などのパターンマッチを直感的に表現することができます．

The Programming Language Egison

Satoshi Egi
University of Tokyo

Egison is a pure functional programming language. The feature of Egison is the strong pattern match facility. With Egison, you can represent pattern matching for data types whose data have no canonical forms, such as set, multiset, or algebraic structures such as group, ring, field, and so on.

1 はじめに

パターンマッチについて考えることは，思っているより非常に重要なことです．プログラミング言語が洗練され，多くの冗長性が排除されてきたが，それでもまだ冗長性が一番目立っている部分は，データの分解の処理、つまりパターンマッチの表現です．よくある例としては，集合などのような順序の関係ないコレクションを扱う際，いちいち順序を持つコレクションであるリストとして捉え直して扱わなければならないというものがあります．これは一般的にいうと，正規形を持たないデータに対してパターンマッチが直接的にできないというようにいえます．正規形というのは，同じデータに対する 1 つの標準的な表現のことです．集合は，正規形を持たないデータ型のもっとも知られているわかりやすい例です．例えば，$\{a, b, c\}$ という集合は，$\{b, a, c\}$ や $\{a, a, c, b\}$ という形でも表せます．ソートして小さい順に並べたものを 1 つの標準的な表現を決めることができそうです．しかし，集合の要素にいつでも順序関係があるとは限りません．また，そもそもソートできた場合でも，それによってパターンマッチが便利になるのはごく一部のパターンだけです．多くの場合，パターンマッチの方法に制限がかかり，パターンマッチの記述は煩雑のままです．

このようにパターンマッチが上手く表現できないデータがある原因は，データの構成，分解の方法に単純なものしか用意されていないことです．データを構成する方法は，決まった有限のデータを受け取るように定義されたデータコンストラクタにデータを渡すという単純なものだけであるし，データを解釈する方法は，その単純な定義に基づいてそれぞれの要素にアクセスするというものだけです．データの構成する方法や解釈する方法は，C 言語の構造体の考え方から未だにほとんど進化していません．

アルゴリズムの表現において，データの解釈の表現，つまりはデータの分解の表現は大きな部位を占めます．それにも関わらず，大雑把なデータの構成，分解の方法しかプリミティブに用意されていないせいで，多種多様なデータ分解に必要な一般的な処理を使いやすい形でモジュール化することができません．

Egison はそこを解決したプログラミング言語です．Egison では，例えば集合やマルチセットなどリスト以外の正規形を持たないデータに対してでも行うことができる一般的なパターンマッチの方法を記述できます．それにより，ほぼ全てのアルゴリズムの表現が Egison で記述することによって簡潔になります．

2 Egison 解説

2.1 4 種類の括弧

Egison には 4 種類の括弧があります．

’(’, ’)’ 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ます．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ります．

’<’, ’>’ で囲まれた式は，データコンストラクタの適用を表します．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ります．

’<’, ’>’ 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ます．パターンコンストラクタはデータコンストラクタと呼び方を変えただけで，同じものです．後で説明します．

’[’, ’]’ で囲まれた式はタプルを表しています．Egison のタプルは多値として使うことができます．1 つの式しか含んでいないタプルはその 1 つの式と同値になります．

```
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) を表します．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 種類があります．’@’ がついていない式は，全体のコレクションの要素の 1 つとして扱われます．’@’ 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*) として扱われます．

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

2.2 組み込みデータ

文字，数，浮動小数点数は組み込みデータとして言語に組み込まれています．

シングルクォートで 1 文字囲むとその文字は文字リテラルとなります．

```
’a’
’b’
’c’
...
```

ダブルクォートで囲まれた文字列は文字列リテラルとなります．文字列は文字のリストとして扱われます．

```
‘‘abc’’
‘‘cde\n’’
...
```

数だけからなる文字列は整数リテラルとなります．

```
1
0
-100
...
```

. が数の間にあると浮動小数点数リテラルとなります．

```
1.0
0.0
-100.012001
...
```

2.3 let と lambda

lambda 構文は，1 つ目の引数にパターン変数 (*pattern variable*) のタプルをとります．パターン変数というのは，先頭に'$' がついている文字列のことです．これからそこで束縛される変数を表します．2 つ目の引数に関数適用した際に実行する式をとります．関数適用の式には，apply 構文を用いたものと， 適用する関数と引数を並べて，'(' と ')' によって囲ったものがあります，apply 構文は，1 つ目の引数に適用する関数をとります．2 つ目の引数に適用する式を書きます．適用する関数と引数を並べて，'(' と ')' によってで囲った式は， apply 構文の糖衣構文です．

```
> (define $f (lambda [$x $y] [(+ x y) (* x y)]))
f
> (test (f 2 4))
[6 8]
> (test (apply f [2 4]))
[6 8]
> (test (apply f (f 2 4)))
[14 48]
```

let 構文は，1 つ目の引数にパターン変数と式のタプルのコレクションをとります．その式を評価した値をパターン変数に束縛します．Egison の let は，相互再帰的な束縛をします．その束縛を現環境追加して，2 つ目の引数の式を実行します．

```
> (test (let {[$f (lambda [$x] (+ (g x) 10))]
              [$g (lambda [$x] (+ x 1))]}
         (f 0)))
11
```

2.4 パターンマッチ

パターンマッチを行うコードを動かしてみましょう．この節では，そのなかでもわかりやすいコレクションのパターンマッチを書いてみます．この節に出てくるコードを実行するには，lib/base.egi と，lib/number.egi，lib/collection.egi をロードする必要があります．load 式はトップ式で，1 つ目の引数で指定されたファイルの内容を読み込みます．

```
> (load ‘‘/path/to/lib/base.egi’’)
> (load ‘‘/path/to/lib/number.egi’’)
> (load ‘‘/path/to/lib/collection.egi’’)
```

では，パターンマッチを行うコードを動かしてみましょう．

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

match 構文は 1 つ目の引数にパターンマッチのターゲット (*target*) 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ります．この式は，{2 7 7 2 7} というコレクションを，(Multiset Integer) としてパターンマッチするという意味になります．(Multiset Integer) という式は，整数のマルチセットという意味です．(マルチセットというのは，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す．) マッチ節は，パターン (*pattern*) とパターンマッチが成功した際に実行される式からなるタプルとして表されます．match 式が実行される際，先頭のマッチ節から順に，パターンマッチに成功するかどうかみていきます．パターンマッチに成功したパターンがあれば，パターンマッチによって計算された束縛フレーム (*binding*

frame) を現環境に追加して，その節の式を評価します．Egison では，パターンマッチによる束縛フレームが複数ある場合があります．その場合，複数ある束縛フレームの 1 つが選ばれます．(仕様上ではどの束縛を選んでもよいことになっているのですが，先頭の束縛フレームを選ぶように実装しています．) 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します．

Egison ではパターンの左側から順にパターンマッチしていきます．Egison のパターンマッチは non-left-linear になっています．non-left-linear は，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という意味です．

’,’ が先頭についたパターンは，バリューパターン (*value pattern*) です．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します．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できます．パターンマッチは，常にパターンの左側から順に行われていきます．

’!’ が先頭についたパターンは，カットパターン (*cut pattern*) です．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ます．それまでのパターンマッチで複数のマッチ可能な候補があっても，そのうち 1 つだけを残してパターンマッチを続けます．この場合は，5 枚中同じ数のカードが 3 枚あるのは 1 パターンしかないので，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のでそれを表現しています．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します．

’_’ はワイルドカード (*wild card*) であり，何に対しもパターンマッチ成功します．Egison では，パターンはファーストクラスオブジェクトです．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ます．

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

次は，パターンマッチによる束縛フレームが複数あって面白い例をみてみましょう．match-all 構文は，match 構文と同じく 1 つ目の引数にターゲットを，2 つ目の引数に型をとります．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ります．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，得られた束縛フレームのコレクションの全ての束縛フレームについて， 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します．

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

> (test (match-all {1 2 3} (Multiset Integer)
          [<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} {}]}

> (test (match-all {1 2 3} (Set Integer)
          [<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}]}
```

パターンコンストラクタ join は引数を 2 つ取ります．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します．パターンコンストラクタ毎のパターンマッチの方法の詳しい記述は型の定義のところで記述されます．次節で型定義について概説します．

他にも例をいくつかあげておきます．

```
> (test (match {5 2 1 3 4} (Multiset Integer)
          {[<cons $n
             <cons ,(- n 1)
              <cons ,(- n 2)
               <cons ,(- n 3)
                <cons ,(- n 4)
                 <nil>>>>>>
            <ok>]
           [ _ <ko>]}))
<ok>
> (test (match-all {1 2 3} (List Integer)
          [<join $hs <cons $x $ts>> [hs x ts]]))
{[{} 1 {2 3}] [{1} 2 {3}] [{1 2} 3 {}]}
> (test (match-all {1 2 3 4 5} (Multiset Integer)
          [<cons $n $rest> [m rest]]))
{[1 {2 3 4 5}] [2 {1 3 4 5}] [3 {1 2 4 5}]
 [4 {1 2 3 5}] [5 {1 2 3 4}]}
```

2.5 型の定義方法

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す．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します．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ます．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す．

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

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します．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します．

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ます．例えば，(Multiset Integer) は数の多重集合という型になり，(Multiset (Multiset Integer)) は数の多重集合の多重集合という型になります．

型は type 式を用いて定義されます．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ります．let 式と同じく相互再帰的な値の定義を行うことができます．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ます．

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ます．この 3 つの関数を使って，マッチ関数 (*match function*) が構成されます．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す．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ます．

var-match 関数，inductive-match 関数，equal?関数についてそれぞれ説明します．

var-match 関数は，パターンが変数パターン (*variable pattern*) である場合に使われます．変数パターンとは，パターン変数だけからなるパターンのことをいいます．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ります．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ます．

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ます．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す．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します．

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ます．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します．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ます．この定義では，equal?関数の中で再帰的に Multiset が呼び出されています．

では，再び inductive-match 関数について説明します．inductive-match 関数は，deconstructor 式を用いて定義します．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ります．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ります．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ます．デコンストラクトマッチ節の 2 つ目の要素に [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ます．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ます．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ります．例の場合だと，型が (Multiset Integer) 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ます．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します．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きます．これのそれぞれ 1 つ目の要素を Integer として，

2 つ目の要素を (Multiset Integer) として， 再帰的にパターンマッチを行われます．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ません．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ます．

3 関連研究

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ります．

そのなかでもっとも有名な研究に Views[5] があります．複数の捉え方が可能な抽象データに対して，複数の方法でパターンマッチできるようにしたものです．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ます．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す．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ます．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ありません．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です．

また，目的が同じ研究に Active Patterns[1][3] があります．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ます．この論文には，多重集合をパターンマッチする例があります．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 [2]．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ません．そのため，Active Patterns には，パターンの先頭から順番に値を確定していくパターンしかパターンマッチすることができません．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です．

First Class Patterns[4] も，Views を拡張した研究です．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ます．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ます．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，その情報をもとにパターンマッチングを行います．First Class Pattern は Haskell の拡張として設計されています．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します．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ます．

4 シンポジウムでの質疑とその後

Q. Egison でパターンマッチを記述することによって本来よりも計算量が大きくなることがあるか？

A. 同じアルゴリズムでパターンマッチを行うのであれば，Egison で記述しても同じ計算量でパターンマッチが行われます．

Q. Egison では効率のよいパターンマッチが簡単に表現できるのか？例えば，要素の数 n で要素が連番となるコレクションをパターンマッチする場合，計算量 *nlog(n)* でパターンマッチできるか？

A. 計算量 *nlog(n)* でパターンマッチを行うには，コレクションの要素をソートしてリストとしてパターンマッチを行うしか Egison でもありません．Egison はひとにとっては単純であるにも関わらず，従来のプログラム言語で書くと煩雑になっていたデータ分解の表現を簡略に書くのに役に立ちます．しかし，ひとにとっても単純でないパターンマッチのアルゴリズムは，単純に記述することができません．

Q. 将棋やリバーシのパターンマッチを簡潔に書くにはどうするのか？

A. 現状の Egison では，まだこれらの表現を簡潔に書くことが難しいです．今回，コレクションに対して行ったような方法をハッシュについても行い，ハッシュデータのパターンマッチも簡潔に表現できるような機構を考えたいと考えています．

参考文献

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

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

[3] 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.

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

[5] 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.