日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

コレクションデータに対してのパターンマッチの実現

江木 聡志

任意個の要素の組み合わせからなるデータ (例えば，リストや集合，多重集合など) をコレクションデータ (collection data) と呼ぶ．集合や多重集合などの概念は，多くの数学的なアルゴリズムにおいて重要であることが多い．しかし，一般的にリスト以外のコレクションデータは正規形を持たないため，強力なパターンマッチを実現するのが難しい．現在まで様々なパターンマッチの拡張が考えられてきたが，そのどの拡張も，正規形を持たないコレクションデータに対しての強力なパターンマッチは実現できていない．本研究では，このような正規形を持たないコレクションデータについても，他のデータ型と同じくらい強力なパターンマッチを実現するための方法と実装を考えた．

1 はじめに

要素の数がいくつでも，リストはリストであるし，集合は集合である．このように任意個の要素からなる抽象データはコレクションデータ (collection data) と呼ばれている．このような種類のデータを一般的な抽象データと区別し，特別なパターンマッチの定義の方法を与えるのが，本研究のアイデアである．

集合や多重集合などといったコレクションデータはほとんどのアルゴリズムに登場し，その場合，それらのデータに対するパターンマッチングやデコンストラクタも必ず登場する．しかし，これらのコレクションデータのパターンマッチのアルゴリズムは直感的には当たり前であるが，手数が多いため，何の用意もなくプログラムに記述すると煩雑になってしまうことが多い．もし，コレクションデータに対するパターンマッチ (パターンマッチはデコンストラクタとしても使用可能) の直感的な記述方法があれば，アルゴリズムの直感的な記述を大きく助ける．

しかし，コレクションデータに対してのパターンマッチングの実現は難しい．その理由は，一般的にリスト以外のコレクションデータは正規形を持たないところにある．例えば，$\{a,b,c\}$ という集合に対して $\{b,a,c\}$ や $\{c,b,a\}$，$\{a,a,b,c\}$ という集合も同値であり，一つの決まった正規形を持たない．このように決まった正規形を持たないデータについてのパターンマッチを行うためにはバックトラッキングを行うことが必要になるため，単純な仕組みで強力なパターンマッチを実現するのは難しい．

従来の方法では，コレクションデータは，他の抽象データと区別されず，型の直和と直積，再帰を用いて定義され，その定義にもとづくパターンマッチの方法が定義されていた．しかし，その方法では，コレクションデータに対しても，直感的なパターンマッチを実現できない．本研究では，コレクションデータ専用のパターンマッチの方法の定義方法を新たに加えることによって，直感的なパターンマッチを実現した．次章以降その方法について説明する．

2 構文の定義とプログラムの例

2.1 マッチ式

本論文では，以下のようなプログラミング言語を考える．

```
<exp> ::= <con>                    (定数)
```

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

```
        | <var>                          (変数)
        | (collect <c-exp> ...)
        | (<exp> <exp>)                  (関数適用)
        | (lambda (<var> ...) <exp>)     (関数定義)
        | (let ((<var> <exp>) ...)
            <exp>)                       (let 式)
        | <match-exp>

<c-exp> ::= <exp>
          | .<exp>
```

collect はコレクションデータのコンストラクタである．このコンストラクタは任意個の引数をとり，それらを組み合わせたコレクションデータを返す．collect は. が頭に付いた式を引数にとることができる．“.” の後ろの式は必ずコレクションデータでなければならない．“.” を使うと “.” の後ろの式が表すコレクションデータの要素を全て含むコレクションデータを作ることができる．このようなあるコレクションデータの一部となっているコレクションデータのことをサブデータ (subdata) と呼ぶことにする．以下は collect の簡単な使用例である．左辺の式は右辺の数の連なりからなるコレクションデータに解釈される．

```
(collect 1 2 3) => {1 2 3}
(collect 1 2 .(collect 3 4) 5) => {1 2 3 4 5}
```

問題のマッチ式の構文の定義は以下のようになる．

```
<match-exp> ::=
   (match <target> <match-function>
      (<condition> <exp>)
      ...)

<target> ::= <exp>

<condition> ::=
   (<pattern> (<rule> ...))

<pattern> ::= <var>
            | _
            | ,<exp>
            | (<pat-exp> ...)

<pat-exp> ::= <pattern>
            | .<pattern>

<rule> ::= (= <var> (<var> ...) <exp>)
         | (: <var> (<var> ...) <exp>)
         | (? <var> (<var> ...) <exp>)
```

ユーザーはマッチ式でパターンマッチの方法をマッチ関数 (match function) により指定する．マッチ関数は，パターンマッチの条件 (condition) とパターンマッチのターゲット (target) を引数に取り，マッチ可能かどうかとマッチ可能な場合は束縛を返す関数である．マッチ関数については後で詳しく説明する．

パターンマッチの条件は，パターンと，ルールのリストからなる．

パターンはパターン変数以外にもワイルドカードをとることができる．ワイルドカードはどのような値ともマッチする．パターンは先頭に “,” のついた式をとることができる．パターン変数とワイルドカードを合わせて変数パターン (variable pattern) と呼ぶことにする.. この “,” のついた式は評価され，その値をパターンマッチする．このようなパターンは定数パターン (constant pattern) と呼ぶ．また，パターンのコレクションもパターンである．このようなパターンはコレクションパターン (collection pattern) と呼ぶ．collect と同じように，パターンはその要素に頭に “.” のついたデータをとることができる．“.” のあとに来たデータは，パターンのターゲットのサブデータにマッチする．このようにパターンはその要素に再帰的にパターンをとりうるので，集合の集合のようなコレクションデータのコレクションデータをパターンマッチすることもできる．

ルールには 3 種類の指定方法がある．一つ目の要素でその方法を指定する．“=” と，“:”，“?” の三種類の指定ができる．二つ目の要素で制限したいパターン変数を指定する．三つ目の要素で，二つ目の要素で指定した制限したいパターン変数の制限を記述するのに必要な別のパターン変数のリストを指定する．四つ目の要素で具体的な制限を記述する．一つ目の要素で “=” を指定した場合は，評価したら定数の値が返ってくる式を記述する．二つ目の要素で指定したパターン変数のとりうる値はその値に制限される．一つ目の要素で “:” を指定した場合は，評価したらコレクションデータが返ってくる式を記述する．二つ目の要素で指定したパターン変数のとりうる値はそのコレクションデータの要素に制限される．一つ目の要素で “?” を指定した場合は，評価したら一項述語が返ってくる式を記述する．二つ目の要素で指定したパターン変数のとりうる値はその一項述語を満たす値に制限される．

以下ルールの具体例である．

```
(= n (x y) (+ x y))
```

このルールの意味は，パターン変数 n は別のパターン変数 x と y に依存し，その値は x+y であるという意味である．

```
(: n () (collect 1 2 3))
```

このルールの意味は，パターン変数 n のとりうる値は 1 か 2，3 のどれかであるという意味である．

```
(? n () even?)
```

このルールの意味は，パターン変数 n の even?という述語を満たす要素であるという意味である．

あとはプログラムの具体例を用いて説明する．以下はユークリッドの互除法のアルゴリズムを，上記の言語を用いて記述したプログラムである．

```
(define gcd
  (lambda (Ns)
    (match Ns (of Set Int)
      (((m ,0) ()) m)
      (((m .Rest)
        ((= m () (min (remove Ns 0)))))
       (gcd (collect m .(map (lambda (n)
                               (modulo n m))
                             Rest)))))))
```

(of Set Int) は整数の集合をパターンマッチするためのマッチ関数である．一つ目のマッチ節の条件部ではルールが指定されていない．この条件では，Ns を集合として捉えたとき，Ns が 0 以外の要素をちょうど一つもつ場合，一つ目のマッチ節でマッチする．二つ目のマッチ節の条件部では，m にどのような値がマッチすべきかルールで指定されている．m は Ns から 0 を除いた要素のうち最小の数が束縛される．Rest は頭に”.” が付いているのでサブデータである．Rest には，Ns から m を除いたコレクションデータが束縛される．マッチ節は上から順にチェックされる．上のほうにマッチする節が優先される．

より複雑なパターンマッチの例として，ポーカーの役判定をするプログラムを示す．このプログラムは要素としてトランプのカードをもつコレクションデータを引数として受け取り，返り値として役の名前を返す．

```
(define poker_hands
  (lambda (Ks)
    (match Ks (of Multiset (** Mark Mod13))
      ((((S ,10) (S ,11) (S ,12) (S ,13) (S ,1))
        ())
       "Royal Straight Flush")
      ((((S n) (S n+1) (S n+2) (S n+3) (S n+4))
        ((= n+1 (n) (+ n 1))
         (= n+2 (n) (+ n 2))
         (= n+3 (n) (+ n 3))
         (= n+4 (n) (+ n 4))))
       "Straight Flush")
      ((((_ n) (_ n) (_ n) (_ n) _)
        ())
       "Four of Kind")
      ((((_ m) (_ m) (_ m) (_ n) (_ n))
        ())
       "Full House")
      ((((S _) (S _) (S _) (S _) (S _))
        ())
       "Flush")
      ((((_ n) (_ n+1) (_ n+2) (_ n+3) (_ n+4))
        ((= n+1 (n) (+ n 1))
         (= n+2 (n) (+ n 2))
         (= n+3 (n) (+ n 3))
         (= n+4 (n) (+ n 4))))
       "Straight")
      ((((_ n) (_ n) (_ n) _ _)
        ())
       "Three of Kind")
      ((((_ m) (_ m) (_ n) (_ n) _)
        ())
       "Two Pair")
      ((((_ n) (_ n) _ _ _)
        ())
       "One Pair")
      (((_ _ _ _ _)
        ())
       "Nothing"))))
```

マッチ関数として，(of Multiset (** Mark Mod13)) が指定されている．これは，マークと 13 の剰余環の直積の多重集合のマッチ関数という意味である．トランプのカードは，マークと 13 の剰余環の直積として表現されている．“**” についてはすぐあとで説明する．マークと数の組み合わせは，プログラムでは要素の数が二つに固定されたコレクションデータとして表現されている．ストレートのパターンマッチにあらわれるパターン変数 n+x は，パターン変数 n に依存している．n+x に束縛されるべき値がどのような値なのかを n なしに説明することができない．このような関係をルールで表現することができる．また，例のように一つのパターンのうちに，同じパターン変数を何度も登場させることができる．

“**” は，複数のマッチ関数を受け取り，それぞれのマッチの対象となるデータを組み合わせたデータ

に対するパターンマッチを行うマッチ関数を返す関数である．例えば (** Int String) は整数と文字列の組み合わせをパターンマッチするためのマッチ関数である．

また，“**” によく似た関数に “||” もある．“||” は，複数のマッチ関数を受け取り，それぞれのマッチの対象となるデータのどれかであるデータに対するパターンマッチを行うマッチ関数を返す関数である．例えば (|| Int String) は整数か文字列であるデータに対するパターンマッチするためのマッチ関数である．

例えば二分木のマッチ関数は以下のようになる．

```
(define Bintree (lambda (inner-type)
  (|| inner-type
      (** inner-type
          (Bintree inner-type)
          (Bintree inner-type)))))
```

Bintree はマッチ関数を受け取り，そのマッチ関数のパターンマッチの対象のデータについての二分木に対するパターンマッチをするためのマッチ関数を返す関数である．整数の二分木のマッチ関数は，(of Bintree Int) とすれば，作成できる．of の定義はこうである．

```
(define of
  (lambda (t .Ts)
     (apply t Ts)))
```

このようにマッチ関数さえ用意されていれば，プログラマーはパターンとルールを記述するだけであらゆるパターンマッチングを実行することができる．しかし，一般的にコレクションデータについてのマッチ関数は複雑でありその記述は容易ではない．したがって本研究では，コレクションデータについてのマッチ関数を定義を簡単にできる仕組みを考え用意した．その仕組みについて次節で説明する．

2.2 マッチ関数の定義

本節ではコレクションデータのマッチ関数の定義の方法について説明する．

まず，コレクションデータのパターンマッチングをどのようなアルゴリズムで行うのか考える．

コレクションデータのマッチ関数は，コレクションパターンのそれぞれの要素の値を確定していき，全ての要素の値を確定することによりパターンマッチを行う．コレクションパターンの要素に定数パターンが含まれているときは，値がその定数に制限されている変数パターンとして扱うのですべての要素が変数パターンと考えてよい．

値の確定していない全てのコレクションパターンの要素のとりうる値の候補を計算する．この要素のとりうる値の候補の集合のことをマッチ域 (match region) と呼ぶ．一つずつ要素の値を確定いき，全ての要素の値を確定すればパターンマッチ完了である．一つ要素の値を確定するたびに，他のすべてのまだ確定していない要素のマッチ域を計算し，とりうる候補の数の少ない要素から確定していく．要素の取りうる値にいくつかの候補がある場合は，ある場合は暫定的にそのどれか一つの候補に確定する．このような他の値の候補を残して暫定的に確定された要素を暫定要素 (temporary fixed element) と呼ぶことにする．また，それに対して，一つしかない値の候補に確定された要素を真確定要素 (completely fixed element) と呼ぶことにする．要素のうち取りうる値の候補がない要素のある場合，かつ確定要素に暫定要素が一つもなくすべて真確定要素である場合はパターンマッチに失敗したことになる．もし確定要素に暫定要素がある場合は，最後に暫定した要素について別の候補で試す．

このパターンマッチのアルゴリズムは，人間がコレクションデータのパターンマッチングを行う際，当たり前に行われるアルゴリズムとほとんど同じである．人間はとりうる候補の少ないコレクションデータの要素からではなく，少なそうなコレクションパターンの要素から確定していくが，あとはまったく同じ方法をパターンマッチを行っている．

この方法で全ての種類のコレクションデータのパターンマッチングを行う．コレクションデータの種類によって変わるのは，コレクションパターンの要素のマッチ域の計算アルゴリズムの部分だけである．したがって，コレクションデータのマッチ関数を定義する際，ユーザーはそれぞれのパターン変数のマッチ域の計算アルゴリズムの部分だけ記述すればよい．以下，マッチ域の計算アルゴリズムについて詳しく考えていく．

コレクションパターンの要素には次の三つのパターンがある．一つ目は，未確定の要素のパターン (unfixed element pattern)，二つ目は，未確定のサブデータのパターン (unfixed subdata pattern)，三つ目は，確定済みの要素のパターン (fixed element pattern) である．確定済みのサブデータのパターン (fixed subdata pattern) は，いくつかの確定済みの要素のパターンと同一視できるのでここでは考えない．このうち未確定の要素のパターンと未確定のサブデータのパターンについてマッチ域を計算する．

コレクションパターン中の未確定のパターンのマッチ域を計算するには，コレクションパターンをリストとしてみて，その未確定のパターンより前にあるパターンのリストと，そのパターン自身と，そのパターンの後ろのパターンのリストの組み合わせをパターンマッチングする．そして，そこから得た情報をもとに未確定のパターンの動く範囲を計算する．ここで計算したコレクションパターンの未確定のパターンの動く範囲を可動域 (movable region) と呼ぶ．可動域と，未確定のパターンにあらわれるパターン変数に対してルールで記述されている制限を照らし合わせてマッチ域を求める．

コレクションデータの種類によって変わるのは，マッチ域の計算のうちでも，コレクションパターンの要素の可動域の計算アルゴリズムの部分だけである．したがって，ユーザーはそれぞれのパターン変数の可動域の計算アルゴリズムを記述すればよい．

以下は多重集合のマッチ関数の定義である．

```
(define Multiset
  (match-function (pattern target inner-type)
    (let* ((cs (filter fixed? pattern))
           (rs (remove-all inner-type
                           target cs)))
      (movable-region pattern
        ((((.Fs1) u (.Fs2))
          ((? Fs1 () all-fixed?)
           (? u () unfixed?)
           (? Fs2 () all-fixed?)))
         (if (= (length rs) 1)
             (collect .rs)
             (collect)))
        ((((._) u (._))
          ((? u () unfixed?)))
         (collect .rs))
        (((((.Fs1) S (.Fs2)))
          ((? Fs1 () all-fixed?)
           (? S () subdata?)
           (? Fs2 () all-fixed?)))
         (collect rs))
        (((((.FSs1) S (.FSs2)))
          ((? Fs1 () all-fixed-or-subdata?)
           (? S () subdata?)
           (? Fs2 () all-fixed-or-subdata?)))
         (collect .(submultisets rs)))))))
```

マッチ関数は，match-function 構文を用いて定義する．match-function は，可動域計算関数を受け取り，マッチ関数を生成する．movable-region 構文を用いて可動域計算関数を定義する．movable-region 構文は可動域を計算するためのマッチ節のリストを受け取る．このマッチ節は前節で定義したものと同じものである．

この例では，それぞれのコレクションパターンの要素の可動域の計算に入る前に，それぞれの計算の中で使う値を計算している．cs には，すでに値が確定したパターンの値のコレクションが束縛されている．rs には，ターゲットから cs を抜いた要素，つまり，まだマッチされていないターゲットの要素のコレクションが束縛されている．

一つ目のマッチ節では，他の全ての要素が確定済みで，可動域計算対象の要素だけが未確定の要素パターンである場合にマッチする．この場合，まだマッチされていないターゲットの要素がただ一つ残っている場合はその要素のみが候補となので，その要素のみを含むコレクションデータを返す．そうでない場合は，候補となる要素がないので，空のコレクションデータを返す．

二つ目のマッチ節では，可動域計算対象の要素が未確定の要素パターンである場合にマッチする．他の要素についての指定はない．この場合は，まだマッチされていないターゲットの要素の集合が，可動域であるので，その集合からなるコレクションデータを返す．

三つ目のマッチ節では，他の全ての要素が確定済みで，可動域計算対象の要素だけが未確定のサブデータパターンである場合にマッチする．この場合は，まだマッチされていないターゲットの要素からなるコレクションデータのみが候補となるので，それのみからなるコレクションデータを返す．

四つ目のマッチ節では，他の全ての要素が確定済み，もしくは未確定のサブデータで，可動域計算対象の要素が未確定のサブデータパターンである場合にマッチする．この場合は，まだマッチされていないターゲットの要素からなる多重集合の部分多重集合の集合が可動域となるので，その集合からなるコレクションデータを返す．

このマッチ関数は，まず全ての未確定の要素のパターンのとる値を確定したあとで，未確定のサブデータのパターンのとる値を確定していく．

集合のマッチ関数は以下のように定義できる．

```
(define Set
  (match-function (pattern target inner-type)
    (let* ((as (unique inner-type target)
               (cs (unique inner-type
                           (filter fixed? pattern)))
               (rs (remove-all inner-type
                               as cs))))
      (movable-region pattern
        ((((.Fs1) u (.Fs2))
          ((? Fs1 () all-fixed?)
           (? u () unfixed?)
           (? Fs2 () all-fixed?)))
         (case (length rs)
           ((0) (collect .as))
           ((1) (collect .rs))
           (else (collect))))
        ((((.As1) u (.As2))
          ((? u () unfixed?)))
         (collect .(append rs cs)))
        (((((.Fs1) S (.Fs2)))
          ((? Fs1 () all-fixed?)
           (? S () subdata?)
           (? Fs2 () all-fixed?)))
         (collect rs))
        (((((.FSs1) S (.FSs2)))
          ((? Fs1 () all-fixed-or-subdata?)
           (? S () subdata?)
           (? Fs2 () all-fixed-or-subdata?)))
         (collect .(subsets as)))))))
```

3 関連研究との比較

過去にも，パターンマッチの拡張の方法がいくつも研究されている．

有名なパターンマッチの拡張に Views [3] がある．

抽象データには複数の捉え方が可能なものがある．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，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．

Views は集合や多重集合のような正規形をもたないデータのパターンマッチには役にたつものではない．

また，本研究と非常によく似た研究に Active Patterns [1] [2] がある．

Active Patterns も本研究と同じく，抽象データに対するパターンマッチの方法をプログラマが指定することができる．しかし，本論文の提案したパターンマッチに比べて，Active Patterns の能力は大きく限られている．

Active Patterns では，本論文で提唱したようなコレクションデータ専用のコンストラクタはなく，コレクションデータは一般的なプログラミング言語と同じように，直積と直和，再帰を用いて定義されていてその再帰的定義に基づくパターンマッチのアルゴリズムを定義する．それゆえ，パターンの先頭から順番にパターンマッチングしていかなければならないという制限をもつ．

また，Active Patterns は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．したがって，集合や多重集合のコレクションデータのパターンマッチにおいて，Active Patterns では，パターン変数がパターンの末尾にしかあらわれることができない．

したがって，コレクションデータに対しては，本論文で提案したパターンマッチよりも Acitive Patterns のパターンマッチの能力は非力である．

4 まとめと今後の課題

コレクションデータ一般のパターンマッチのアルゴリズムを抽象化することは，アルゴリズムの直感的な表現を考える上でかなり重要な意味を持っている．本論文はその抽象化の一つを提案した．この提案の主な特徴は，コレクションデータ専用のコンストラクタを用意したところと，バックトラッキングを組み込んだ専用のデコンストラクタ (マッチ関数) を記述する仕組みを用意したところにあった．これらのアイデアによってコレクションデータに対するパターンマッチ

の能力は従来のものにくらべてかなり強力になった．例えば，ポーカーの役判定プログラムなどは劇的に簡単に記述できるようになっている．

しかし，本研究のパターンマッチにもまだ制限はある．例えば，パターン変数間の同値関係以外の相互依存関係を記述できない．その例としては，集合のパターンマッチで，全てのパターン変数のとる値は互いに異なるというルールをそのまま記述することができない．

また，マッチ関数の記述法にもまだ議論の余地はある．本研究が提案したものよりも良い記述方法があるかもしれない．

参 考 文 献

[1] M. Erwig. Active patterns. *Implementation of Functional Languages*, LNCS 1268:21–40, 1997.

[2] 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.

[3] 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.