# 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
東京大学

2011/9/29

# 既存の言語の問題点

- 既存のプログラミング言語では集合やマルチセットなど正規形を持たないデータのパターンマッチを自然に表現できない
  - 正規形とは同じデータに対する1つの標準的な表現のこと
  - 既存の言語では，集合やマルチセットなどをパターンマッチする際，いちいちリストに変換してパターンマッチしなければならない
- 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一般的な方法をモジュール化する方法が必要！！

# プログラミング言語Egison

- 純粋関数型言語
- 超強力なデータデコンストラクタ

(data deconstructor)

- インタプリタをHaskellで実装している
- Hackageのパッケージとして配布している

Haskell Platformがインストールされていれば,
% cabal install egison
というコマンド1つでインストールできる

パターンマッチするターゲット

パターンマッチする型

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

パターン

パターンマッチに成功したら実行する式

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

Match-allは全てのマッチする組み合わせを計算する

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

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

リストとして
パターンマッチ

```
(test (match-all {1 2 3} (Multiset Integer)
        [<cons $x $ts> [x ts]]))

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

マルチセットとして
パターンマッチ

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

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

集合として
パターンマッチ

```
(test (match-all {1 2 3 4} (List Integer)
        [<join _ <cons $m <join _ <cons $n _>>>> [m n]]))
```

コレクションの2つの要素をそれぞれのパターン変数に束縛するパターン

```
=> {[1 2] [1 3] [1 4] [2 3] [2 4] [3 4]}
```

```
(test (match-all {2 8 7 2 7} (Multiset Integer)
        [<cons $m <cons ,m _>> m]))
```

2つ同じ要素を含むコレクションである場合にパターンマッチ

```
=> {2 7}
```

# 主な先行研究


- 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.
- M. Erwig. Active patterns. Implementation of Functional Languages, pages 21–40, 1996.
- M. Tullsen. First Class Patterns. Practical Aspects of Declarative Languages, pages 1–15, 2000.
- S. Antoy. Programming with narrowing: A tutorial. Journal of Symbolic Computation, 45(5):501– 522, 2010.

# Views

- 特徴
  - ほとんどの研究がこの研究を引用している
  - 同じデータを複数の方法でデコンストラクトすることができる
  - 2つのデータ型の間の変換方法を定義する
  - 最近Haskellに組み込まれた
- 弱点
  - パターンマッチの際にバックトラックを行うことができない
    - 正規形を持たないデータ型に対するパターンマッチには使えない

# Active Patterns

- 特徴
  - Viewsの拡張
  - コンストラクタ毎にデコンストラクトのアルゴリズムを付加する
  - この機構を利用してグラフのパターンマッチをする方法を論じた論文もある
- 弱点
  - バックトラックできず，パターンの先頭からパターンマッチするしかない
    - 単純なパターンしか扱えない

# Narrowing

- 特徴
  - 論理型プログラミングのユニフィケーションを応用したパターンマッチ
  - Curryというプログラミング言語に組み込まれている
  - パターンマッチの際にバックトラックを行える
- 弱点
  - 型ごとにパターンマッチの方法を定義する方法がない
    - パターンマッチの式の記述が直感的ではない

# First Class Patterns

- 特徴
  - コンストラクタ毎にデコンストラクトの方法を付加する
  - パターンマッチの際にバックトラックが行える
  - パターンがファーストクラスオブジェクト
  - 提案機構がHaskellの拡張として提案されている
- 弱点
  - パターンの表現力が低い
    - 人間にとっては簡単なパターンでも，簡潔に表現できないパターンがある
    - 明らかに無駄な探索を行わないように簡潔に書けない

# Egison

- 特徴
  - 型ごとにデコンストラクトの方法を定義する
  - パターンマッチの際の探索を簡潔な表現で制御できる

# Egisonの解説

- 型の定義方法
- パターンの表現を簡略化するための工夫
- 複雑なパターンの記述について
- パターンマッチの計算量

# Egisonの解説

- 型の定義方法
- パターンの表現を簡略化するための工夫
- 複雑なパターンの記述について
- パターンマッチの計算量

# Egisonの型の定義

- パターンとデータを引数に取り，全ての可能な束縛を計算するマッチ関数なるものを型ごとにユーザは記述する
  - 例．Multisetのマッチ関数は．パターン<cons $x $xs>，ターゲット{1 2 3}の場合，以下のような結果を返す
    - `{[x 1] [xs {2 3}]},{[x 2] [xs {1 3}]},`
      `{[x 3] [xs {1 2}]}`
- マッチ関数の記述のために以下の3つの関数を定義する
  - Var-match: パターンが変数パターンである場合にパターンマッチを行う関数
  - Inductive-match: パターンが帰納的に構成されたパターンである場合にパターンマッチを行う関数
  - Equal?: パターンが値である場合にパターンマッチを行う関数

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

Multisetは型aを受け取って型を返す関数

(Multiset Int) : 整数のマルチセット
(Multiset (Multiset Int)) :
　整数のマルチセットのマルチセット

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

パターンコンストラクタnilは引数を取らない

1つ目の引数は型aとして，2つ目は(Multiset a)として再帰的にパターンマッチ

1つ目のと2つ目の引数を両方とも(Multiset a)として再帰的にパターンマッチ

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

ターゲットが{1 2 3}である場合

{}を返す

{[1 {2 3}] [2 {1 3}] [3 {1 2}]}を返す

{[{} {1 2 3}] [{1} {2 3}] [{2} {1 3}]
{[3 {1 2}] [{1 2} 3] [{1 3} 2] [{2 3}
1] [{1 2 3} {}]}を返す

```
(test (match-all {1 2 3} (Multiset Integer)
        [<cons $x $xs> [x xs]]))

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

1つの要素と残りのコレクションとに分割するパターンマッチ

```
(test (match-all {1 2 3} (Multiset Integer)
        [<join $xs $ys> [xs ys]]))

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

2つのコレクションに分割するパターンマッチ

# Egisonの解説

- 型の定義方法
- パターンの表現を簡略化するための工夫
- 複雑なパターンの記述について
- パターンマッチの計算量

# 表現を簡略化するための工夫

- どの型でパターンマッチを行うのかを指定してパターンマッチを行う
  - ネストしたデータ型のパターンマッチが簡潔になる
  - 同じコンストラクタが似たデータ型で再利用できる
- 同じパターンの内の変数に束縛された値を他の部分をパターンマッチする際に参照できる
- パターンマッチの際に行われる探索を簡潔に制御する仕組みがある

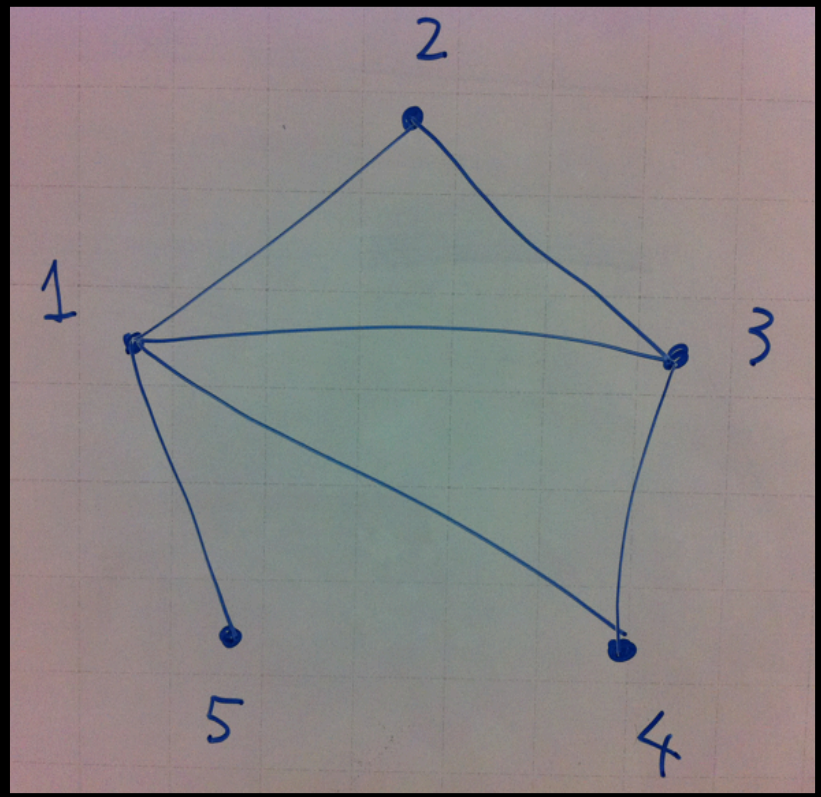
```
(test (match {2 7 7 2 7} (Multiset Integer)
        {[<cons $m
           <cons ,m
            <cons ,m
             <cons $n
              !<cons ,n
                 <nil>>>>>>
          <ok>]
         [_ <ko>]}))

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

'!' は Prolog のカットと同じ.
それまでの探索で複数の候補が
あってもその1つに絞る.

上のプログラムは下のプログラムと同値

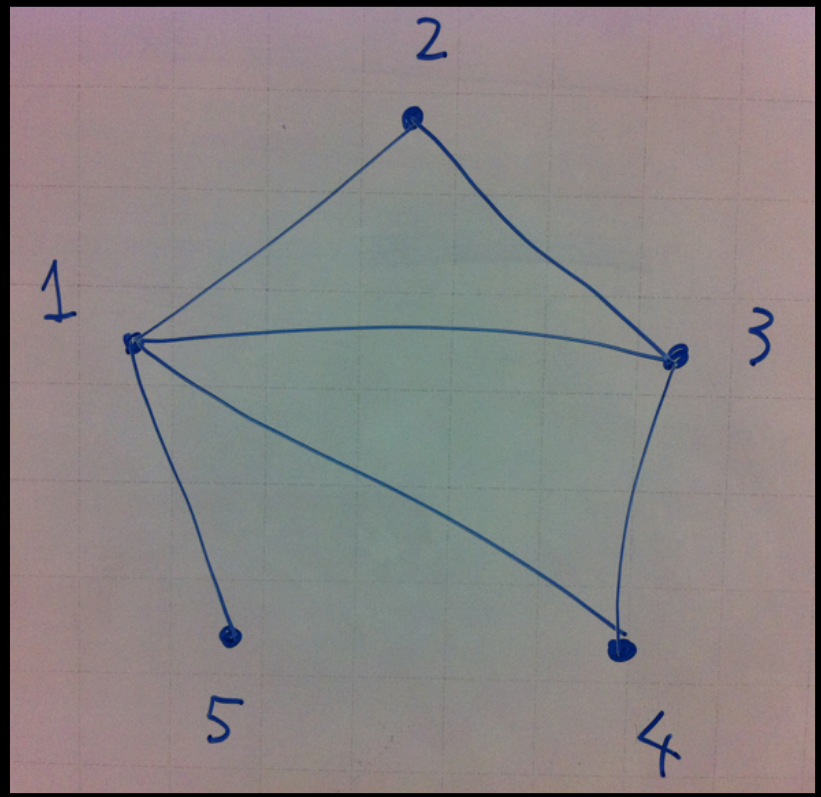
```
(define $Node Integer)

(define $NodeInfo
  (type
    {[$var-match (lambda [$tgt] {tgt})]
     [$inductive-match
      (deconstructor
        {[node [(Multiset Node) Node (Multiset Node)]
          {[<node $in $n $out> {[in n out]}]}]})]
     [$equal? (lambda [$val $tgt]
                (match [val tgt] [NodeInfo NodeInfo]
                  {[[<node $in $n $out>
                     <node ,in ,n ,out>]
                    <true>]
                   [[_ _] <false>]}))]}))

(define $Graph (List NodeInfo))
```

```
(define $g {<node {2 3 4 5} 1 {2 3 4 5}>
            <node {1 3} 2 {1 3}>
            <node {1 2 4} 3 {1 2 4}>
            <node {1 3} 4 {1 3}>
            <node {1} 5 {1}>})
```



```
(test (match-all g Graph
        [<cons <node _ $n1 <cons $n2 _>>
          <join _
           <cons <node _ ,n2 <cons $n3 _>>
            <join _
             <cons <node _ ,n3 <cons ,n1 _>>
              _>>>>>
         [n1 n2 n3]]))
```

3つの辺からなる閉路を探す
パターンマッチ




```
=> {[1 2 3] [1 3 4]}
```

# Egisonの解説

- 型の定義方法
- パターンの表現を簡略化するための工夫
- 複雑なパターンの記述について
- パターンマッチの計算量

# 複雑なパターンの記述

- パターンがファーストクラスオブジェクトであるため，パターンを引数にとり，パターンを返す関数が書ける
- 正規表現に対応するようなパターンも記述可能

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

=> <ok>
```

'1'だけからなるコレクションに
パターンマッチ

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

=> <ok>
```

1つ以上の'1'からなるコレクションに
パターンマッチ

# Egisonの解説

- 型の定義方法
- パターンの表現を簡略化するための工夫
- 複雑なパターンの記述について
- パターンマッチの計算量

# Egisonのパターンマッチの効率

- FAQ
  - 自動でパターンマッチの際の探索が行われる
  - 既存の言語と同じように効率の良いようにプログラムが書けるのか？？
- Answer
  - 明らかに無駄な探索は行われないように簡潔に表現することはできる
  - 本質的に複雑なアルゴリズムが簡潔な表現で書けるようになるというわけではない

```
(define $loop
  (lambda [$k]
    (of {<nil> <cons ,k (loop (- k 1))>})))
```

(k，k-1，k-2，…)と続くパターン

```
(define $isStraight?
  (lambda [$Ns]
    (match Ns (Multiset Integer)
      {[<cons $n (loop (- n 1))> <ok>]
       [_ <ko>]})))
```

計算量のオーダは Length(Ns)^2

```
(define $isStraight?
  (lambda [$Ns]
    (match (qsort Ns) (List Integer)
      {[<cons $n (loop (- n 1))> <ok>]
       [_ <ko>]})))
```

計算量のオーダは
Length(Ns)*log(Length(Ns))

オーダーLength(Ns)*log(Length(Ns))でパターンマッチするには，
ソートしてリストとしてパターンマッチするしかない

# まとめ

- パターンの表現力が向上したことにより，正規形のデータ型に対するパターンマッチが簡潔に表現できるようになった
  - 型ごとにパターンマッチの方法を記述
  - パターンの左側の変数に束縛された値を参照できる
  - カットパターンによる探索の制御
- 特にコレクションのパターンマッチについては直感のままに表現できるようになった
  - 例．ポーカーの役を判定するパターンマッチ

# 今後の方針

- パターンマッチのさらなる拡張
  - 連想配列のようなデータ構造に対しても自然なパターンマッチを実現する
- Egisonを使って色々なプログラムを書く
- コンパイラや豊富なライブラリを作り，もっと実用に耐えうる言語を目指す

# Egisonのドキュメント

- http://hagi.is.s.u-tokyo.ac.jp/~egi/egison/index-j.html
  - プログラミングマニュアルが置いてあります
- https://github.com/egisatoshi/egison
  - GitHubでソースが公開されています