Functional Programing in Pattern-Match-Oriented Programming Style

Satoshi Egi (Rakuten Institute of Technology, Rakuten Inc. / The University of Tokyo)
Yuichi Nishiwaki (The University of Tokyo)

What is Egison and PMOP?

Egison is a programming language that we have developed to advocate pattern-match-oriented programming (PMOP). Egison features user-extensible non-linear pattern matching with backtracking.
PMOP confines recursions that describe backtracking into non-deterministic patterns.

Traditional FP (Haskell)

```
delete _ [] = []
delete x (y : ys) | x == y = ys
delete x (y : ys) = y : delete x ys
```

PMOP (Egison)

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

Example. Poker hand

We can describe patterns for a multiset. Users can define a pattern-match method for multisets.

```
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"

poker [Card Spade 5, Card Spade 6, Card Spade 7, Card Spade 8, Card Spade 9]
-- "Straight flush"

poker [Card Spade 5, Card Diamond 5, Card Spade 7, Card Club 5, Card Heart 7]
-- "Full house"

poker [Card Spade 5, Card Diamond 10, Card Spade 7, Card Club 5, Card Club 8]
-- "One pair"
```

Example. Twin primes

The combination of non-linear patterns and backtracking is powerful.

Twin primes

```
take 5 (matchAll primes as list integer with
   | _ ++ $p :: #(p + 2) :: _ -> (p, p + 2))
--  [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31)]
```

Sequential prime pairs that are not twin primes

!pat: not-pattern
pat1 & *pat2*: and-pattern

```
take 5 (matchAll primes as list integer with
   | _ ++ $p :: (!#(p + 2) & $q) :: _ -> (p, q))
-- [(2, 3), (7, 11), (13, 17), (19, 23), (23, 29)]
```

Prime pairs whose form is (p, p+6)

```
take 5 (matchAll primes as list integer with
 | _ ++ $p :: _ ++ #(p + 6) :: _ -> (p, p + 6))
-- [(5, 11), (7, 13), (11, 17), (13, 19), (17, 23)]
```

Prime triplets

```
take 5 (matchAll primes as list integer with
 | _ ++ $p :: $q :: #(p + 6) :: _ -> (p, q, p + 6))
-- [(5, 7, 11), (7, 11, 13), (11, 13, 17), (13, 17, 19), (17, 19, 23)]
```

Example. Davis-Putnum algorithm

PMOP allows programmers to focus on writing the essential parts of an algorithm by distinguishing two types of computations:

1. Computations that can be implemented in backtracking algorithms (***backtrack-able computations***);
2. Computations that are essential for improving the time complexity of an algorithm for solving a problem (***essential computations***).

Traditional FP

```
let rec dp clauses =
 if clauses = [] then true else if mem [] clauses then false else
 try dp (one_literal_rule clauses) with Failure _ ->
 try dp (pure_literal_rule clauses) with Failure _ ->
 dp(resolution_rule clauses);;

let one_literal_rule clauses =
 let u = hd (find (fun cl -> length cl = 1) clauses) in
 assignTrue u clauses;;

let pure_literal_rule clauses =
 let us = unions clauses in
 let u = hd (find (\u -> mem (negate u) us) us) in
 assignTrue u clauses;;
```

Backtrack-able computation
Backtrack-able computation
Essential computation
Backtrack-able computation
Essential computation
Backtrack-able computation
Backtrack-able computation
Essential computation
Backtrack-able computation

Traditional FP mixes two computations.

OCaml program taken and modified from [Harrison, 2009]

PMOP (Egison)

```
def dp cnf :=
 matchDFS cnf as multiset (multiset integer) with
 | [] -> True
 | [] :: _ -> False
 -- one-literal rule
 | ($x :: []) :: _ -> dp (assignTrue x cnf)
 -- pure literal rule
 | ($x :: _) :: !((#(negate x) ::  _) :: _) -> dp (assignTrue x cnf)
  -- otherwise
 | _ -> dp (resolution cnf)
```

Backtrack-able computations

Essential computations

PMOP distinguishes two computations.

PMOP quizzes

We can redefine various list functions in PMOP style.

```
def member x xs := match xs as list eq with
 | (1) -> True
 | _ -> False

member 2 [1, 2, 3] -- True
member 4 [1, 2, 3] -- False
```

```
def deleteAll x xs := matchAll xs as list eq with
 | (2) -> y

deleteAll 2 [1, 2, 3, 2] -- [1, 3]
```

```
def unique xs := matchAllDFS xs as list eq with
 | _ ++ $x :: (3) -> x

unique [1, 2, 3, 2] -- [1, 3, 2]
```

```
def intersect xs ys := matchAll (xs, ys) as (set eq, set eq) with
 | ($x :: _, (4) ) -> x

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

```
def difference xs ys := matchAll (xs, ys) as (set eq, set eq) with
 | ($x :: _, (5) ) -> x

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

Answers: (1) _ ++ #x :: _, (2) _ ++ (!x & $y) :: _, (3) !(_ ++ #x :: _), (4) #x :: _, (5) !(#x :: _)