Recursive Pattern Matching on Concrete Data Types
with appendix

Pierre Jouvelot
CAI, Ecole des Mines de Paris, Fontainebleau (F.)
LCS, Massachusetts Institute of Technology, Cambridge (U.S.A.)
(E-mail: `jouvelot@{fremp11.bitnet,brokaw.lcs.mit.edu}`)

Babak Dehbonei
Corporate Research Center, BULL, Louveciennes (F.)
(E-mail: `Babak.Dehbonei@crg.bull.fr`)

ACM SIGPLAN Notices, Vol 24, Issue 11, Nov 1989, pp 84–91

1 Introduction

The functional programming model favors recursive definitions of functions by structural induction over user-defined concrete data types. Most functional languages (e.g., Lisp, ML, Miranda, Haskell) offer powerful programming constructs that ease the writing of programs based on this idea; destructuring `let`, `multiple-value-bind`,...

We propose an extension to the usual notion of pattern-matching, called *Recursive Pattern Matching*. Our motivation is based on the observation that most real-life data types are recursive, e.g. abstract syntax trees, control graphs, ... Consequently, many functions that manipulate these data types are recursive. The RPM technique combines pattern-based dispatching and recursive function calls on subcomponents of complex values.

This technique has been successfully used on a large scale inside the Velour project. This vectorizing and parallelizing compiler is 11,000 LeLisp lines long and most of its modules use recursive pattern matching on abstract syntax trees. Compared to a first version of Velour that didn't use recursive pattern matching, we observed a 20 to 30 per cent decrease[1] in the size of the code, without significant performance penalty. Like any structuring concept, we also noticed that its usage entailed a substantial reduction in the number of design and coding errors.

In the remainder of this paper, we survey the related work (section 2), define precisely the notion of recursive pattern matching (section 3), give a set of simple examples (section 4), investigate further improvements (section 5) and conclude (section 6). An appendix provides a complete CommonLISP implementation of RPM.

2 Related Work

Pattern matching is an important facility provided by many modern programming languages. Basically, once a given data type (seen in this framework as a set of functions that allow the manipulation of values of this type) has been defined, patterns (i.e., expressions that involve data type functions and unbound variables) can appear as left hand sides of any binding construct; the net effect is to bind the variables to the corresponding values in the right hand side. The right hand side value has to match the corresponding pattern.

[1] This imprecision comes from the fact that the second version of Velour also introduced numerous enhancements that influenced the code size.

The simplest way of using this technique is given in many widely used imperative languages. In Pascal [JW85], the `case` command allows the programmer to execute statements according to the value of an integer or an enumerated variable. In the C programming language [KR78], an equivalent treatment is performed on integer expressions using the `switch` statement. However, these matching possibilities are not sufficiently powerful to be applied on more abstract data.

In Lisp-like languages [St84], a flavor of abstract pattern matching is present for structured lists or trees. This is performed on function calls or macro expansions with the dotted notation or the CommonLISP keywords (e.g., `&rest`, `&optional`).

A more sophisticated technique is used in the ML functional language [M83]. It is performed on expressions that have (possibly recursive) sum types over a set of available types. An example will clarify this approach:

```
#rectype Num =
#        zero |
#        succ of Num ;;
```

The recursive `Num` type denotes the set of positive or null integers. We can then define a function `Add` that computes the sum of two values (given in a pair argument) of type `Num`.

```
#letrec sum = fun
#       (zero,Y). Y |
#       (succ X,Y). sum (X, succ Y) ;;
```

The matching is performed on the two type constructors `zero` and `succ` of `Num`; this is usually implemented by a one-way unification routine.

The mechanism of *views* has been proposed for matching expression that have an abstract data type [W87]. This allows the programmer to use pattern-matching without losing the abstraction. Changing the implementation of an abstract data type won't require any modification of programs that import this type (this wouldn't be the case with the ML kind of matching which is concrete). An external (resp. internal) description of an abstract value is given through the attribute *out* (resp. *in*) in the data type definition. For instance, an integer can be seen as:

```
view int ::= Zero | Succ int
     in n         = Zero if n=0
                  = Succ(n-1) if n > 0
     out Zero     = 0
     out (Succ n) = n+1
```

In this example, the abstract definition of an integer has a concrete description that is the corresponding integer number. The pattern matching is performed on this concrete version of the data type via the abstract functions.

A very similar notion to our pattern facility has been recently introduced in [J87]. This approach is based on the attribute grammar paradigm applied to LML, a lazy version of ML. A recursive control structure called `case rec` is introduced to traverse concrete data structures and compute attributes; this construct is complex (not restricted to non-circular grammars) and was not implemented at that time. Furthermore, it relies heavily on the laziness of LML since it doesn't restrict the attributes computation in the way our construct does.

Several other studies have also been made in the context of actor languages. These languages define pattern matching operators as higher level objects. Backtracking is also allowed in this scheme to accomodate constraints that might be defined on patterns.

3 Recursive Pattern Matching

Recursive pattern matching is a language-independant notion. It can be used in any functional language that supports structured concrete data types.

3.1 Type Notation

A concrete type `T` can either be a basic type (like `Int` or `Bool`) or a constructed type. A constructed type is a sum of product types (i.e., disjoint unions of structures with multiple members like `t1:T1 + t2:(T2 × T3)`). We assume that each value of sum type `T` with tag `t` satisfies the run-time predicate has_tag_t; this means that tags (representing types) are carried around by the underlying implementation. For every sum type `T`, the function *untag* retrieves the untagged component of the value. For every product type `T` that has `T1` as component, there exists a function `T_T1` that retrieves the component of type `T1` in an object of type `T`.

3.2 Definition of RPM

The core of the RPM technique is a special form, called `rpm`. Its BNF syntax is the following:

`rpm` ::= `(rpm` root Clause*`)`
root ::= Expression
Clause ::= `(`Pattern body `)`
Pattern ::= `(`head member*`)`
head ::= Identifier
member ::= Identifier
body ::= Expression*

where *item** represents a non-empty list of *item*, Identifier is a Lisp symbol and Expression any Lisp expression.

Recursive pattern matching performs a recursive traversal of a tree-shaped concrete data type. The fundamental characteristic of `rpm` is to hide the recursive invocations inside the very definition of the patterns. Namely, whenever a member appears in a clause, then its value in the corresponding body is the result of the recursive call on the member instead of the more usual member value.

The informal semantics of an `rpm` expression is the following. First, the root expression is evaluated; its value v is a structured value of type t which has to be one of the different heads of the list of clauses. Each clause with pattern p is successively checked to see whether t matches p, i.e. the head h of p equals t. The first clause c that succeeds is chosen (if none, an error is reported). The result of the evaluation of the list of expressions inside c in an augmented environment is returned. The augmented environment is defined as follows: h is bound to $(untag v)$ and each member m is bound to the result of the recursive evaluation of the whole process on the value `(`h_m `(untag`v`))` where h_m is the function that retrieves the m component of a value of type h.

The formal semantics of `rpm` is given below with a denotational flavor; without loss of generality, we used a restricted version of `rpm` that limits the number of clauses to two and allows only one member in a pattern and one expression in each clause. The standard direct semantics $\mathcal{E}$ of any functional language is extended with the semantics of `rpm`. It uses a function $\mathcal{C}$ that takes two continuations: the first one is used once a pattern matches with the value and the second one allows the trial of subsequent clauses (or failure). The semantics of the pattern matching process is the function $\mathcal{P}$. We give below the types of these functions:

$$\mathcal{E} : \text{Expression} \to Store \to Result$$
$$\mathcal{C} : \text{Clause} \times \text{Clause} \to Value \to Cont \to Cont \to Store \to Result$$
$$\mathcal{P} : \text{Pattern} \to Value \to (\text{Identifier} \times \text{Identifier} \times Value)$$
$$Cont : Value \to Store \to Result$$
$$Result : (Value + \mathbf{error})$$

The semantic equations are the following:

$\mathcal{E}[\![$ `rpm E C1 C2` $]\!]s = \mathcal{C}[\![$ `C1 C2` $]\!](\mathcal{E}[\![$ `E` $]\!]s)(\lambda vs.v)(\lambda vs.\mathbf{error})s$

$\mathcal{C}[\![\ \texttt{C1 C2}\ ]\!]vk_1k_2s = \mathcal{C}[\![\ \texttt{C1}\ ]\!]v(\lambda vs.\mathcal{C}[\![\ \texttt{C1 C2}\ ]\!]vk_1k_2s)$
$(\lambda vs.\mathcal{C}[\![\ \texttt{C2}\ ]\!]v(\lambda vs.\mathcal{C}[\![\ \texttt{C1 C2}\ ]\!]vk_1k_2s)(\lambda vs.\textbf{error}))s$
$\mathcal{C}[\![\ \texttt{P E}\ ]\!]vk_1k_2s = \text{let } \texttt{Id1}, \texttt{Id2}, a = \mathcal{P}[\![\ \texttt{P}\ ]\!] \text{ in}$
$\text{if } has\_tag_{Id1}(v) \text{ then } \mathcal{E}[\![\ \texttt{E}\ ]\!][untag\ v/\texttt{Id1}][k_1(a(untag\ v))s/\texttt{Id2}]s \text{ else } k_2vs$
$\mathcal{P}[\![\ \texttt{(Id1 Id2)}\ ]\!] = \texttt{Id1}, \texttt{Id2}, \texttt{Id1\_Id2}$

where `Id1_Id2` is the function that returns the `Id2` member of a value of type `Id1`. Note the recursive behavior of `rpm` pictured in the use of $\mathcal{C}[\![\ \texttt{C1 C2}\ ]\!]$ in its own definition.

3.3 RPM Computes Primitive Recursive Functions

The purpose of this section is to prove that the RPM technique has the expressive power of primitive recursion.

Definition [C87]. A function f of arity n is defined by *primitive recursion* over the functions g and h if and only if:

- $f(x_1, ..., x_n) = g(x_1, ..., x_n)$ if $x_n = 0$,
- $h(x_1, ..., x_n, f(x_1, ..., x_n - 1)$ otherwise

Theorem *Any primitive recursive function f can be encoded with* ***rpm*** *without recursion.*

Proof. Let `Num` be the recursive concrete type `Zero + Succ:Num`. Let f' be the curried function such that $f'(x_n)(x_1, .., x_{n-1}) = f(x_1, ..., x_n)$.

By definition, if x_n is `Zero` we have:

$$f'(x_n) = \lambda x_1, .., x_{n-1}.g(x_1, .., x_n)$$

and if x_n is the successor of y (i.e., $y = \texttt{Num-Succ}\ x_n$):

$$f'(x_n) = \lambda x_1, .., x_{n-1}.h(x_1, ..., x_n, f'(y)(x_1, ..., x_{n-1}))$$

Thus, by definition of `rpm`:

$f'(x_n) = \texttt{rpm}\ x_n$
$((\texttt{Zero})\ \lambda x_1, .., x_{n-1}.g(x_1, .., x_n))$
$((\texttt{Succ Num})\ \lambda x_1, .., x_{n-1}.h(x_1, ..., \texttt{Succ}, \texttt{Num}(x_1, ..., x_{n-1})))$

Inside the body of the second clause, `Succ` is bound to the current value of x_n and `Num` denotes the result of the recursive invocation of f' on y if $y = \texttt{Num-Succ}\ x_n$. We can define f in the following way:

$f(x_1, ..., x_n) = (\texttt{rpm}\ x_n$
$((\texttt{Zero})\ \lambda x_1, .., x_{n-1}.g(x_1, .., x_n))$
$((\texttt{Succ Num})\ \lambda x_1, .., x_{n-1}.h(x_1, ..., \texttt{Succ}, \texttt{Num}(x_1, ..., x_{n-1})))$
$x_1...x_{n-1})$

The formal proof of equivalence between these two forms is straightforward and left to the reader. □

Note that we supposed that concrete types were not circular. If we relax this restriction (i.e., we quit the pure functional paradigm) or allow abstract types instead of concrete types, then partial recursion can be encoded with `rpm`. The idea is to use the potentially infinite traversal on a circular value to implement the minimization operator of partial recursion. Recall that any recursive function (i.e., partial recursive) can be coded with composition of functions, projections, primitive recursion and minimization. Thus, `rpm` codes for any recursive function.

3.4 RPM with CPS

RPM allows the recursive definition of expressions where more than one argument is needed. The trick to adapt `rpm` to this requirement is to use higher-order functions in a way reminiscent of the Continuation Passing Style [St77]. Basically, each clause returns a function that is abstracted over these arguments and it is up to the caller (i.e., the previous clause in the dynamic call sequence) to provide the appropriate parameter values. This can also be used each time a clause treatment requires some inherited values from its caller.

The very definition of `rpm` is bottom-up and using this "CPS"-like programming paradigm is a way to simulate a top-down behavior. We found this twist very successful in our practical experiments and easy to use (once assimilated by programmers !).

4 Examples

In the sequel, we will use the CommonLISP notation; product types are coded by `defstruct`, sum types by a corresponding `deftype` with an `or` constructor and basic types by the underlying implementation. We present three simple examples that use the `rpm` package provided in the Appendix: a factorial function, the computation of the free variables of a λ-expression and a Lambda-Calculus evaluator.

4.1 Factorial

The computation of the factorial of an integer number is an elementary example where the power of our `rpm` function is shown. Let us first define the type of integer numbers and some functions on them:

```
(defsum num zero succ)
(defproduct zero)
(defproduct succ
  (of :type num))

(defconstant num-1 (make-succ :of (make-zero)))

(declare (function num-product (num num) num))
```

where `num` is either a `zero` or a `succ` with a unique member `of` which is a `num`. For convenience, we defined `num-1` and declared the product function on `nums` (the code of which is left as an exercise to the reader).

The `rpm` version of factorial on values of type `num` is the following:

```
(defun factorial (num)
  (rpm num
       ((zero) num-1)
       ((succ of) (num-product succ of))))
```

which can be used in the following way :

```
-> (fact (make-succ :of (make-succ :of (make-zero))))

#(succ #(succ #(zero)))
```

Note that unlike other programming techniques, there is no explicit recursive call to `factorial`; it is embedded inside the `rpm` macro.

4.2 Free Variables of Lambda Expressions

The Lambda-Calculus manipulates λ-expressions. Its syntax is :

```
(defsum lambda-expression variable application abstraction)
(defproduct variable
  (name :type string))
(defproduct application
  (operator :type lambda-expression)
  (operand :type lambda-expression))
(defproduct abstraction
  (variable :type variable)
  (body :type lambda-expression))

(declare (function variable= (variable variable) t))
```

A λ-expression is either a variable, an application of two λ-expressions or an anonymous function with a bound variable as formal argument and a λ-expression as body (abstraction). We introduce the notion of a free variable in a λ-expression by structural induction on the domain of λ-expressions:

Definition [S77] A variable `x` *occurs free* in a λ-expression `E` if and only if:

- `E` is `x`,
- `E` is `(`E_1 E_2`)` and `x` occurs free in E_1 or E_2
- `E` is `(lambda V E)`, `x` and `V` are different and `x` occurs free in `E`

We define the function that computes the set of all free variables present in a λ-expression:

```
(defun free-variables (expression)
  (rpm expression
       ((variable) `(,variable))
       ((application operand operator)
         `(,@operand ,@operator))
       ((abstraction body)
         (remove (abstraction-variable abstraction) body
                 :test #'variable=))))
```

Each clause returns a list of variables after applying the appropriate treatment (e.g., removing the bound variable from the list of free variables of the body of an abstraction).

4.3 A Lambda-Calculus Evaluator

We will use the previous data type and associated functions to write a simple evaluator of λ-expressions. This will be the occasion of using higher order functions as a means to deal with arguments; the `evaluate` function takes, beside the expression, a `store` that maps variables to `num` values (for instance).

We used the `[` macro character to wrap the special form `funcall` around the arguments (see Appendix). They wouldn't be required if we used a one-namespace language like Scheme.

```
(defun update-store (value variable store)
  #'(lambda (v)
      (if (variable= v variable)
          value
        [store v])))
```

```
(defun evaluate (expression store)
  [(rpm expression
        ((variable)
         #'(lambda (store) [store variable]))
        ((application operand operator)
         #'(lambda (store)
             [[operator store] [operand store]]))
        ((abstraction body)
         #'(lambda (store)
             #'(lambda (value)
                 [body (update-store value
                                     (abstraction-variable abstraction)
                                     store)])))))
   store])
```

The function `update-store` updates the store (which is a function) to bind the value to the variable. The constant `init-store` denotes an empty store. The core function `evaluate` evaluates an expression in a given store. The key idea is that each clause in the `rpm` macro returns a function that maps stores to values. We give below a simple example that evaluates `((lambda (x) x) 1)` in an empty initial store:

```
-> (let* ((x (make-variable :name "x"))
          (y (make-variable :name "y"))
          (f (make-abstraction :variable x :body x)))
     (evaluate (make-application :operator f :operand y)
               (update-store num-1 y #'(lambda (variable) :unbound))))

#(succ #(zero))
```

5 Future Work

There are many improvements that can be added to this current definition of the `rpm` special form:

- multiple values could be returned by `rpm`. Actually the extended version of `rpm` used in the Velour project allows such an extension.
- more general patterns that are not limited to direct subcomponents could be introduced. For instance, one might want to perform the recursive calls on specific deeper subtrees instead of being limited to just a one-level recursion. The implementation provided in the Appendix includes a limited version of this idea; if a subcomponent on which a recursive call has to be performed happens to be a list, the recursive process is mapped on each element of the list and a list of results is returned. This could also be applied to vectors.
- a default pattern could be introduced (*a la* _ of ML) to trap the values that didn't match any of the clauses.

6 Conclusion

Recursive Pattern Matching is a programming technique that combines the advantages of simple pattern matching on structured values and recursive function definition. It has been widely used in the development of a vectorizing compiler prototype and proved quite powerful and useful.

A portable CommonLISP implementation of `rpm` is provided.

Acknowledgements

We would like to thank Vincent Dornic for pointing out the quite relevant paper [J87].

References

[C87] Cohen, D.E *Computability and Logic.* Ellis Horwood, Halstead Press, J.Wiley and Sons, New York 1987

[JW85] Jensen, K., and Wirth, N. *The Pascal User Manual and Report.* Third Edition, Springer-Verlag, New York 1985

[J87] Johnsson, T. Attribute Grammars as a Functional Programming Paradigm. In the *Proceedings of the 1987 Int. Conf. on Func. Prog. Lang. and Comp. Arch.*, Portland, 1987

[M83] Milner, R. A Proposal for Standard ML. *Polymorphism Letters I*, Bell Labs, 1983.

[KR78] Kernighan, B., and Ritchie, D. *The C Programming Language.* Prentice-Hall, 1978.

[S77] Stoy, J. E. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[St84] Steele, G. L., Jr. *CommonLISP: The Language.* Digital Press, 1984.

[St77] Steele, G.L., Jr. *Rabbit.* MS Thesis, MIT AI Lab, 1977

[W87] Wadler, P. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the Conference on Principles Of Programming Languages.* ACM, Munich, 1987.

Appendix: A CommonLISP Implementation of rpm

```
;; The "rpm" (Recursive Pattern Matching) macro package.
;;
;; IMPORTANT NOTE: The rpm macro can only be used on CommonLISP structures
;; that are implemented as named vectors. Use the following declaration
;; for that purpose: (defstruct (foo (:type vector) :named) ...). Without
;; this definition, the rpm package may (or may not) work depending on
;; a particular implementation of structures. We provide DEFSUM and
;; DEFPRODUCT macros that introduce these particular declarations.
;;
;; This software is provided without any guarantee. Any trouble can be
;; reported to jouvelot@brokaw.lcs.mit.edu or Babak.Dehbonei@crg.bull.fr.
;;
;; Copyright (C) 1989 - Pierre Jouvelot & Babak Dehbonei.

(provide :rpm)
(in-package :rpm)
(export '(rpm defsum defproduct))

(deftype base-type ()
  "The type of values that are self evaluating."
  '(or symbol integer float character string))

(defmacro defproduct (type &rest domains)
  "TYPE is the product domain of DOMAINS"
  `(defstruct (,type (:type vector) :named) ,@domains))

(defmacro defsum (type &rest domains)
  "TYPE is the sum domain of DOMAINS"
  `(deftype ,type '(or ,@domains)))

;; [ ... ] are used to hide funcalls.
;;
(set-macro-character
 #\[
 #'(lambda (stream char)
       `(funcall ,@(read-delimited-list #\] stream t))))
(set-macro-character #\] (get-macro-character #\) nil))

(defmacro rpm (object &rest clauses)
    "To recurse on the value OBJECT with the recursive pattern matching
CLAUSES, use the following syntax:

      (rpm <exp> ((<type> <member1> ... <membern>) <body>) ...)

Inside <body>, <type> is bound to the current value and <memberi> to
to i-th recursive call result of rpm on <type>-<memberi> on the current
Newgen object. If the member is a list, then a list of results is
returned."
    (let ((loop (gensym))
          (obj (gensym)))
        `(labels ((,loop (,obj)
                         (etypecase
                          ,obj
                          (base-type ,obj)
                          (cons (mapcar #',loop ,obj))
                          (vector
```

```
                          (cond ,@(rpm-clauses obj clauses loop))))))
            (,loop ,object))))

(defun rpm-clauses (obj clauses loop)
  "Tests all the CLAUSES to find the first one that matches the type of
the value OBJ. The recursive function to use for members is LOOP."
    `(,@(mapcar
         #'(lambda (clause)
             (let* ((pattern (car clause))
                    (head (car pattern)))
               `((,(intern
                    (concatenate 'string (symbol-name (intern head)) "-P"))
                  ,obj)
                 (let ((,head ,obj)
                       ,@(recurse-members head obj (cdr pattern) loop))
                   ,@(or (cdr clause)
                         '((declare (ignore ,@pattern))))))))
         clauses)
        (t (error "rpm: unknown value ~D" ,obj))))

(defun rpm-members (head obj members loop)
  "Creates the bindings of each element of MEMBERS to the return of
the recursive call of LOOP on the MEMBER of OBJ (which is of type HEAD)."
    (mapcar #'(lambda (member)
                `(,member
                  (,loop (,(intern
                            (concatenate 'string
                                         (symbol-name (intern head))
                                         "-"
                                         (symbol-name (intern member))))
                          ,obj))))
            members))
```