TOWARDS FUNCTIONAL PROGRAMMING IN PROLOG

Antonio L. Furtado*

Rio Scientific Center
IBM Brasil
Estrada da Canoa, 3520
22.610, Rio de Janeiro, RJ

ABSTRACT

The integration of functional and logic programming is attempted, using the strategy to add a functional component to Prolog. The component takes the form of extended computable expressions, allowing user-defined functions and operators as well as a number of functional forms. The problem of evaluating expressions combining functions and predicates is investigated. Examples are provided to illustrate the discussion. The paper includes a prototype implementation.

1. INTRODUCTION

Two sharply different programming paradigms have been identified [Ba]:

- **Procedural** programming, based on the machine architecture originally conceived by von Neumann;
- **Applicative programming**, oriented towards mathematical computation.

Applicative programming, in turn, may follow different but - so we argue - compatible orientations, depending on the chosen mathematical domain. The LISP language (see [Wi] for a comprehensive presentation) for example, is directed towards **functions**. The APL language [Iv] has a rich set of **operations**. A more recent proposal [BWW] stresses **functional forms**. Perhaps all these languages might be collected under the **functional programming** orientation.

The other significant orientation is **logic programming**, exemplified by Prolog. **Predicates** are the driving mathematical concept in logic programming. Although predicates may be seen as the class of functions returning Boolean values, they are treated differently in Prolog, as compared to LISP for instance. In Prolog, a predicate does not return any value - it **succeeds** or **fails** in the situations corresponding to returning "true" or "false", respectively. Moreover, although the arguments of Prolog predicates may be nested functional terms, these are not submitted to any kind of evaluation.

Some Prolog dialects offer a limited functional capability. IBM Prolog [VM], which closely follows the Edinburgh Prolog de facto standard ([WMSW], page 428), provides **computable expressions**, involving numbers, atoms, strings, the four basic arithmetic operators plus some standard numeric functions (sine, logarithm, etc.) and string-handling functions. The ":=" symbol, not to be confused with procedural assignment, effects the functional evaluation of its right-hand side operand and then tries to unify the result with the left-hand side operand.

In this paper, our objective is to explore the combination of logic and functional programming. This has been recognized as a worthwhile objective [Ro] and several strategies to accomplish it have been proposed [BeL,Bo]. Our strategy is to add a functional programming component to Prolog, essentially by introducing **extended computable expressions**, via a new "<=" operator that generalizes ":=". We claim that the powerful machinery underlying Prolog - namely the **resolution** theorem-proving technique, based in turn on the **unification** algorithm - makes it ideal to host the combined orientations.

The paper is organized as follows. Section 2 introduces the syntax and examples of extended computable expressions containing user-defined functions and operators. Other features of extended computable expressions are treated in the next two sections. Section 3 presents a number of functionals (called elsewhere higher-order functions [Ro], functional forms [Ba,BWW,GJ,Fl] or operations on functions [BrL], page 18). Section 4 investigates how to conciliate the evaluation of functions with that of predicates. For the sake of readability, the presentation is mostly done through examples. Section 5 contains the conclusion. The complete listing of the prototype implementation is given in the Appendix.

---

* On leave from the Pontifícia Universidade Católica do Rio de Janeiro

2. FUNCTIONS AND OPERATORS

2.1. Extended computable expressions

A new operator "<=" has been added, extending the ":=" operator to evaluate computable expressions. Extended computable expressions include (other features will be deferred to sections 3 and 4):

1. lists;
2. predicate terms;
3. conditional expressions;
4. user-defined functions and functional operators.

An example with a conditional expression is:

```
<- read(N) & M <= (ge(N,0) -> N; 0).
```

which unifies M with the value of N if N is non-negative, or with zero otherwise. In general, the value of a conditional expression of the form

```
<- V <= (p1 -> v1; p2 -> v2; ... ; pn -> vn).
```

is the result of evaluating the first vi such that the corresponding logical expression pi is true.

2.2. Defining and using functions

Predicate "addfun" adds functions to the workspace. Actually, it uses "addax" to add duly transformed clauses to implement the functions as predicates, plus "is_function" clauses to declare their nature as well as number of parameters, so that they be recognized and handled correctly by "<=".

```
<-addfun(
factorial(N) <=
  (N == 0 -> 1;
   N * factorial(N - 1))).

<- X < = factorial(5).
```

The call to "addfun" adds the clauses below to the workspace. Note that an extra parameter, V, is included to receive the value that the function would yield. If a function (or operator) will be needed again in later sessions, it may be convenient either to "save" or "bloc_save" it, or to write it directly in predicate format, rather than re-executing "addfun" (or "addop" - section 2.2) at each time.

```
factorial(V,N) <-
  V <= (N == 0 -> 1;
        N * factorial(N - 1)).

is_function(factorial(*)).
```

Other useful examples follow: one numerical (square root) and the others LISP-like.

```
<-addfun(
sqrt(X) <= sqrt1(X,9,1.E-8)).

<-addfun(
sqrt1(X,R,T) <=
  ((D := abs(R - X/R) &
     lt(D,T)) -> R;
   sqrt1(X,(R + X/R)/2.0,T))).

<- X <= sqrt(225).
```

```
<-addfun(
head(X.Y) <= X).

<-addfun(
tail(Z) <=
  (Z == nil -> nil;
   Z = (X.Y) -> Y)).

<-addfun(
cons(X,Y) <=
```

```
(is_list(Y) -> (X.Y);
 true -> (X.Y.nil))).

<-addfun(
subst(X,Y,Z) <=
 ((Y == Z) -> X;
  is_elmt(Z) -> Z;
  true -> cons(subst(X,Y,head(Z)),
               subst(X,Y,tail(Z))))).

<- L = (a.b.a.c.d.a.nil) &
   X <= subst(p,a,L).
```

As happens with predicates, functions do not have to be defined as a single block. The definition of a function f may be done through more than one "addfun", resulting in the addition of an equal number of f predicate clauses. A single "is_function" clause will be created, however.

2.3. Defining and using functional operators

Similarly, there is an "addop" predicate to add functions associated with an operation symbol. An "op" clause must be added separately. To define an operator to raise a number to an integer power, one enters:

```
op("xx",lr,130).

<-addop(
(X xx Y) <=
  (Y = 1 -> X;
   true -> (X * (X xx (Y - 1))) )).

<- B <= 2 xx 3 .
```

We might prefer an upward-pointing arrow instead of "xx", but the choice is limited of course to the symbols available on the keyboard being used.

3. FUNCTIONALS

Higher-order constructs are often useful to avoid explicit control structures. A few have been provided, in order to

- Distribute a binary operator over n arguments, supplied as a list. The symbol denoting distribution is the inverted slash. Examples are:

  ```
  <- X <= ("*"\ (3 . 5 . 6 . nil)).
  <- Y <= (||\('a' . 'b' . 'c' . nil)).
  ```

  which unify X with 90 and Y with 'abc'. The minus and the times operators must be written between single or double quotes.

- Apply lists of extended computable expressions to lists of arguments. Several possibilities are offered by the "map" functionals below:
  - map(e,a) - the (trivial) case of one expression, one argument;
  - map_1n(e,a1.a2. ... .an.nil) - one expression, n arguments
  - map_n1(e1.e2. ... .en.nil,a) - n expressions, one argument
  - map_bi(e1.e2. ... .en.nil,a1.a2. ... .an.nil) - n expressions, n arguments, so that each ei will be applied to the respective ai;
  - map_cart(e1.e2. ... .en.nil,a1.a2. ... .am.nil) - n expressions, m arguments, with each ei being applied to all aj.

  On the other hand, the extended computable expressions used as the first argument of the map functionals offer several possibilities. If they contain one or more variables, all occurrences of the first variable will be instantiated with elements taken from the second argument, according to the type of map. If the extended computable expression consists of a single constant, the constant will be the result; an interesting example is the map to compute the cardinality of a list (see below), where as many ones as there are elements in the list are produced and then counted. The expression may involve conditionals, function

composition, operators, etc. Once their variables are instantiated with values from the second argument, the expression is evaluated by "<=="; however, if any variable remains, there may be problems, which requires some delayed or lazy evaluation device, to be introduced in section 4.

An example is the computation of the standard deviation, defined as the square root of the sum of the squared deviations (between each value and the average) divided by the cardinality minus one.

```
<-addfun(
card(L) <=
  (L = nil -> 0;
  (+\
    map_ln(1,L)))).

<-addfun(
avg(L) <=
  (L = nil -> 0;
   N <= card(L) + 0.0 ->
    (+\ L) / N)).

<-addfun(
std(L) <=
  ((A <= avg(L) & N <= card(L)) ->
       sqrt( (+\
    map_ln(  (*1 - A) xx 2 , L) ) /
            (N - 1.0) ))).

<- L = (8 . 5 . 4 . 12 . 15 . 5 . 7 . nil) &
   Z <= map_n1(card(*1).avg(*1).std(*1).nil, L).
```

The goal computes the cardinality, average and standard deviation of a set of numbers.

- Perform function exponentiation with fixed-point conditions. Function exponentiation, here denoted by "@", is the composition of a function with itself either

  1. a given number of times, or
  2. a variable number of times.

The latter case is useful to find the smallest exponent of a function f for which some condition is true. However, if the condition cannot be met, an infinite loop would occur. If f has a fixed-point, our evaluator is able to check if ever f(X) = X at some step, in which case a failure stops the process. Especially with this provision, function exponentiation is most convenient to express **repetition** [BrL] in a pure functional programming style (by contrast see the "while" construct in [Ba] and its definition in terms of exponents in [Fl]).

Two examples are supplied. In the first, one checks if the atoms a and c appear together somewhere in a list. N will be instantiated with 2 and the expression N + 1 will be equal to 3. If the condition were not met, a loop would not occur because of the fixed-point tail(nil) = nil. Note the use of "<=" as a predicate, to check if the result of evaluating the right-hand side expressions is a and c, respectively.

```
<- L = (a.b.a.c.d.a.nil) &
   a <= head(N@tail(L)) &
   c <= head((N+1)@tail(L)).
```

The second example has an interesting interpretation: given an iterative command in a conventional programming language:

for i := 20 to 10 by -3

it answers the questions:

- how many iterations will normally be executed? (given by I)
- what will be the last value of i? (given by M)

Function decr3 is defined on the natural numbers, whence the fixed-point decr(N) = 0 if N-3 is less than or equal to 0, and decr(0) = 0. Notice that we chose to write function decr3 directly in predicate format.

```
decr3(Z,X) <-
  W := X - 3 &
  (lt(W,0) -> Z = 0;
   Z = W).

is_function(decr3(*)).

<- N = 20 &
   M <= I@decr3(N) & lt(M,10).
```

4. COMBINING PREDICATES AND FUNCTIONS

In functional languages, a "predicate" is a function that returns Boolean values. In Prolog, however, predicates do not return values - they succeed or fail. Further, the arguments of functional language predicates can be computable expressions, a particularly important case being calls to other functions. In Prolog, predicate arguments are not evaluated. Finally, functions (Boolean-valued or not) may be partial, i.e. undefined for certain values of their arguments.

Towards the integration of predicates and functions we provide three main features:

- A prefix operator "$", which, when applied to a predicate causes the evaluation of its arguments, as extended computable expressions, before the predicate itself is evaluated.
- A prefix operator ">", which, when applied to a function, makes the evaluation of the function conditional on all its arguments being fully instantiated (a case of "lazy" evaluation).
- A function "bool", whose argument is a predicate expression. As the expression is evaluated one of the built-in predicate symbols "true" or "fail" is returned, depending on whether or not the evaluation succeeds.

An example previously introduced in section 3 can be slightly rewritten with "$", so that the built-in "eq" predicate can be used instead of "<=".

```
<- L = (a.b.a.c.d.a.nil) &
   $eq(a,head(N@tail(L))) &
   $eq(c,head((N + 1)@tail(L))).
```

Similarly, the "sqrt1" function of section 2.2 becomes:

```
<-addfun(
sqrt1(X,R,T) <=
  ($lt(abs(R - X/R),T) -> R;
   sqrt1(X,(R + X/R)/2.0,T))).
```

The other features are illustrated in the examples below. The first involves a multiple-choice school test consisting of five questions, the correct answers being M = (a,c,b,c,a). One may wish to compare to M the sequence A of the answers supplied by some student. The goal below first yields a sequence C where Ci is "true" if Ai = Mi, and "fail" otherwise. C is produced in two stages: the application of "map_1n" results in the list

```
>bool(eq(a,*)).>bool(eq(c,*)).>bool(eq(b,*)).>bool(eq(c,*)).>bool(eq(a,*)).nil
```

where the ">" operator keeps the member expressions unevaluated in view of the remaining variable, and at the second stage the application of "map_bi" instantiates each second operand variable with the corresponding element of A whereupon the evaluation takes place, giving:

```
true.fail.fail.true.fail.nil
```

Next, C is used in the comparison: (1) within an APL-like "compress" to indicate which answers are correct (in the example, the first and the fourth); (2) in an "or-expression" and in an "and-expression" over the "true" and "fail" predicates, to see whether "at least one answer" and whether "all answers" are correct.

```
<-addfun(
compress(T,L) <=
  (T == nil -> nil;
   true<=(head(T)) ->
     cons(head(L),compress(tail(T),tail(L)));
   true ->
     compress(tail(T),tail(L)))).

<-addfun(
iota(I) <=
  iota(1,I.nil)).
<-addfun(
iota(T,I) <=
  (T.nil == I -> I;
   cons(T,iota(T + 1,I)))).

<- A = (a.b.a.c.b.nil) &
   M = (a.c.b.c.a.nil) &
   C <= map_bi(
         map_ln( > bool(eq(*,*)),M), A) &
   OK <= compress(C,iota(5)) &
   SOME_OK <= bool(| \C) &
   ALL_OK <= bool(& \C).
```

The next example shows how to compute the outer product of two vectors, V1 and V2, to obtain a matrix M, such that Mij = V1i * V2j. The application of "map_ln" produces a list of >(v1i * X) terms, where the v1i are (constant) elements from V1i and X denotes a variable. The " > " operator inhibits the execution of the multiplications until the application of "map_cart" instantiates each second operand variable with each element from V2.

```
<-C <= map_cart(
        map_ln( >(V1 * V2), (1 . 2 . 3 . nil)),
        (4 . 5 . 6 . nil)).
```

Map composition is a powerful and flexible device. Comparing the map composition to compute the standard deviation (section 3) with both map compositions above, one notes that they follow different patterns:

map(.,map(.,.)) - in the former
map(map(.,.),.) - in the latter

As a last example (taken from [BrL]), let us consider again the "decr3" function, defined at the end of section 3. If we now define over the natural numbers a function "g", by

```
<-addfun(
g(Y) <=
  ($eq(I@decr3(Y + 1),1) -> I)).

<- X <= g(12).
<- L <= bool(X <= g(11)).
```

then g(Y) yields Y/3 for all Y that are multiples of 3. Otherwise g(Y) **fails**, due to the fixed-point condition imposed on "decr3". This seems an appropriate way to express, in the extended Prolog, that function "g" is **undefined** for numbers that are not multiples of 3.

5. CONCLUSION

The examples presented along the preceding sections show how to define and use functions and function-based constructs in a functional notation that harmonizes well with the notation used for predicates. Predicates and

functions can thus be combined to solve problems involving both mathematical domains. The prototype implementation demonstrates the feasibility of the strategy.

REFERENCES

[Ba] J. Backus - Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs - Comm. of the ACM - vol. 21, n. 8 (1978) 613-641.

[BeL] M. Bellia and G. Levi - The relation between logic and functional languages: a survey - The Journal of Logic Programming - vol. 3 (1986) 217-236.

[Bo] H. Boley - RELFUN: a relational/functional integration with valued clauses - SIGPLAN Notices, vol.21, n.12 (1986) 87-98.

[BrL] W. S. Brainerd and L. H. Landweber - Theory of Computation - John Wiley (1974).

[BWW] J. Backus, J. H. Williams and E. L. Wimmers - FL language manual (preliminary version) - T.R. RJ 5339 - IBM Almaden Research Center - (1986).

[Fl] A. C. Fleck - Structuring FP-style functional programs - Computer Languages, vol. 11, n. 2 (1986) 55-63.

[GJ] C. Ghezzi and M. Jazayeri - Programming language concepts - John Wiley (1982).

[Iv] K. E. Iverson - A programming language - John Wiley (1962).

[Ro] J. A. Robinson - The future of logic programming - Information processing 86 - H. J. Kugler (ed.) - North-Holland (1986) 219-224.

[VM] VM/Programming in logic - program description and operations manual - doc. IBM SB11-6374-0 (1985).

[Wi] P. H. Winston - LISP - Addison-Wesley (1984).

[WMSW] A. Walker, M. McCord, J. F. Sowa and W. G. Wilson - Knowledge systems and Prolog - Addison-Wesley (1987).

ACKNOWLEDGEMENT

I am grateful to my colleague Ramiro Guerreiro for many suggestions and comments.

APPENDIX

PROGRAM TO IMPLEMENT THE EXTENSION

```
/* special symbols */

op("<=",lr,50).
op("$",prefix,150).
op("@",lr,500).
op("\",rl,33).
op(">",prefix,150).

/* evaluator for                */
/* extended computable expressions */

cexp(X,E) <-
  (is_elmt(E) | is_list(E)) -> X = E;
  is_cond(E) -> c_cond(X,E);
  is_nfunction(E,P,N,A) ->
    nfunct(N,A,X,P);
  is_function(E) -> funct(X,E);
  is_operation(E) -> oper(X,E);
  is_pred(E) -> X = E;
  true ->
    (E =.. (OP . A) &
     f_arg(A,B) &
     F =.. (OP . B) &
     ev(X,F)).

<=(X,E) <- cexp(X,E).

ev(X,F) <-
  (X := F & / |
   X = F).

is_elmt(X) <-
   var(X) | atomic(X) | stringp(X).

is_list(nil).
is_list(*.*).

is_pred(P) <-
   ¬(is_function(P) |
    is_operation(P)) &
   P =.. (OP.*) &
   ¬op(OP,*,*) &
   ¬is_f_predef(OP).

is_f_predef(F) <-
```

```
  on(F,max.min.abs.sin.cos.pi.
     exp.log.len.strip.upper.
     lower.substring.nil).

on(X,(X.Y)).
on(X,(Y.Z)) <- on(X,Z).

is_cond(* -> *).
is_cond(* ; *).

is_nfunction(E,P,N,A) <-
  E =.. ("@".P.R.nil) &
  R =.. (N.A.nil).

c_cond(X,C -> E) <- / &
  C &
  cexp(X,E).
c_cond(X,E1 ; E2) <- / &
  c_cond(X,E1) -> true;
  cexp(X,E2).

funct(X,F) <-
  F =.. (OP.A) &
  ((OP == map_ln | OP == map |
    OP == bool | OP == ">") ->
    (A = (S.AA) &
     f_arg(AA,C) &
     B = (S.C)) ;
   f_arg(A,B)) &
  ((OP == ">") &
    G =.. (wait_no_var.X.B) & / |
   G =.. (OP.X.B)) &
   G.

oper(X,F) <-
  F =.. (OP.A) &
  f_arg(A,B) &
  H =.. (OP.B) &
  G = opr(X,H) &
  G.

f_arg(nil,nil).
f_arg(A.R,A.S) <-
   is_pred(A) & / &
   f_arg(R,S).
f_arg(A.R,B.S) <-
  cexp(B,A) &
  f_arg(R,S).

/* addition of functions and */
/* functional operations     */
/* to the workspace          */

addfun(F <= B) <-
  F =.. (OP.A) &
  G =.. (OP.X.A) &
  arg_patt(A,AA) &
  H =.. (OP.AA) &
  Z = (G <- (X <= B)) &
  addax(Z) &
  (is_function(H) & / |
   addax(is_function(H))).

addop(F <= B) <-
  F =.. (OP.A) &
  arg_patt(A,AA) &
  H =.. (OP.AA) &
  addax(opr(V,F) <- (V <= B)) &
  (is_operation(H) & / |
   addax(is_operation(H))).

arg_patt(X,Y) <-
  compute(list,E,on(*,X),nil,Y).

/* distribution of binary operators */
/* over n arguments                 */

distr(O,L,X) <-
  (stringp(O) -> st_to_at(O,P);
   P = O) &
  distr1(P,L,R.nil) &
  X <= R.

distr1(O, A.B.R, S) <-
  M =.. (O.A.B.nil) &
  distr1(O,M.R,S).

distr1(O,A,A).

opr(X,(O\L)) <-
  X <=
   (distr(O,L,X) -> X).

is_operation(*\*).

\(O,L) <-
  distr(O,L,X) &
  X.

/* application of sets of functions */
/* over sets of arguments           */

map(Y,F,A) <-
  pmap(F,A,X) &
  Y = X.

map_ln(Y,F,A) <-
  pmap_ln(F,A,X) &
  Y = X.

map_n1(Y,F,A) <-
  pmap_n1(F,A,X) &
  Y = X.

map_bi(Y,F,A) <-
  pmap_bi(F,A,X) &
  Y = X.

map_cart(Y,F,A) <-
```

```
  pmap_cart(F,A,X) &
  Y = X.

is_function(map(X,Y)).
is_function(map_1n(X,Y)).
is_function(map_n1(X,Y)).
is_function(map_bi(X,Y)).
is_function(map_cart(X,Y)).

pmap(F,A,X) <-
  listvar(F,nil) & / &
  X <= F.

pmap(F,A,X) <-
  copy(F,G) &
  listvar(G,Y.R) &
  Y <= A &
  X <= G.

pmap_1n(*,nil,nil) <- / .

pmap_1n(F,A.R2,X.R3) <- / &
  copy(F,G) & copy(F,H) &
  pmap(G,A,X) &
  pmap_1n(H,R2,R3).

pmap_1n(F,A,X) <-
  copy(F,G) &
  pmap(G,A,X).

pmap_n1(nil,*,nil) <- / .

pmap_n1(F.R1,A,X.R3) <- / &
  copy(F,G) & copy(R1,H) &
  pmap(G,A,X) &
  pmap_n1(H,A,R3).

pmap_n1(F,A,X) <-
  copy(F,G) &
  pmap(G,A,X).

pmap_bi(nil,nil,nil) <- / .

pmap_bi(F.R1,A.R2,X.R3) <- / &
  copy(F,G) & copy(R1,H) &
  pmap(G,A,X) &
  pmap_bi(H,R2,R3).

pmap_bi(F,A,X) <-
  copy(F,G) &
  pmap(G,A,X).

pmap_cart(nil,*,nil) <- / .

pmap_cart(F.R1,A,X.R3) <- / &
  copy(F,G) & copy(R1,H) &
  pmap_1n(G,A,X) &
  pmap_cart(H,A,R3).

pmap_cart(F,A,X) <-
  copy(F,G) &
  pmap_cart(G,A,X).

/* function exponentiation */

nfunct(F,A,X,N) <-
  (¬ var(N) -> M <= N;
   M = N) &
  copy(M,NN) &
  nfunct1(F,A,X,M,NN).

nfunct1(F,A,X,N,NN) <-
  B <= A &
  (X = B & N = 0 |
   W =.. (F.B.nil) &
   (var(NN) &
    Z <= W &
    ¬ (B == Z) |
    ¬ var(NN)) &
   nfunct1(F,W,Y,M,NN) &
   (¬ var(NN) &
    ge(M,NN) & / & fail |
    true) &
   N := M + 1 &
   X = Y).

/* combination of functions */
/* and predicates           */

pval(P) <-
  P =.. (OP.A) &
  f_arg(A,B) &
  Q =.. (OP.B) &
  Q.

$P <- pval(P).

bool(X,P) <-
  X <=
   (P -> true;
    fail).

is_function(bool(*)).

wait_no_var(Z,E) <-
  ((listvar(E,N) & N == nil) ->
     (Z <= E);
   Z = >(E)).

is_function(>(*)).
```