ITERATION FOR APPLICATIVE LANGUAGES

Antonio L. Furtado and Paulo A. S. Veloso

Departamento de Informatica
Pontificia Universidade Catolica do R.J.
22.453 Rio de Janeiro, R.J.
Brasil

ABSTRACT

The mathematical notion of function exponentiation is used to introduce an iteration construct suitable to applicative languages. The elements of the construct are explained as well as its evaluation. A prototype Prolog implementation is included to illustrate the discussion.

1. INTRODUCTION

The control component of applicative languages should naturally be congenial to the mathematical concepts that underly such languages, generally functions (as in LISP) or predicates (as in Prolog). Both LISP and Prolog support recursion; function composition in LISP roughly corresponds to the conjunction of literals in Prolog. The **if-then-else** and **case** constructs of procedural languages have a powerful counterpart in the **cond** functional of LISP.

However, the convenience of further enriching the control structure of applicative languages has had some recognition. LISP went perhaps too far, incorporating the **prog** feature, whereas many Prolog dialects, going, more justifiably, beyond the strict Horn-clause notation, adopted disjunction and some kind of **case** construct.

Different programming styles result from having or not constructs like these. In pure LISP, it is common to have a function **f** calling other "auxiliary" functions **f1**, **f2**, etc. In many cases the auxiliary functions are never directly invoked, since they are really a part of **f** that the syntax of the language does not allow to write within the expression defining **f** (cf. the "local" functions defined in **where** expressions of CPL [Bar1]).

The difference is even more striking in Prolog. A predicate **p** that is true under more than one conjunction of conditions must be expressed by an equal number of separate clauses in the strict notation; with disjunction and **case**, **p** can often be programmed as a single unit. Whether separate clauses or single units are preferable depends on the nature and intended use of **p**. If **p** is part of a rule-structured knowledge base that is supposed to grow incrementally, separate clauses may be more convenient. But if **p** is a driving or scheduling module of an expert system [FCT], or if it expresses an algorithm, then legibility is usually favored by expressing the entire body of **p** as a single block.

In at least one situation separation remains mandatory: the body of a function or predicate must be interrupted whenever

repetition occurs. The repetitive segment of the program then takes the form of an auxiliary recursive function or predicate, to be called from the point where the repetition should appear. A seemingly more desirable alternative, to be investigated in this paper, would be to provide some in-line **iteration construct**, as most procedural languages do.

2. ITERATION IN VON NEUMANN LANGUAGES

The reluctance to introduce an iteration construct in an applicative language can be understood from the example below. The reader will readily notice that it computes the greatest common divisor of two numbers, here 24 and 9.

```
  ..............
n := 24;
m :=  9;
while m ≠ 0 do
  begin
    temp := n;
    n := m;
    m := temp mod m
  end;
  ..............
```

Variables n and m are first "initialized" and, inside the **while** construct, undergo destructive assignments, which happen to be the von Neumann machine feature most criticized by applicative programming supporters. The "temporary" variable is a particularly unpleasant machine-like feature. However, without it, there would be no natural sequence of statements allowing the new values of both n and m to be computed from their previous values, i.e. the above sequence in fact emulates the parallel assignments

```
[n := m, m := n mod m]
```

Languages like Algol 68 support parallel execution, but still leave us with the destructive assignment problem. Indeed the semantics of iteration seems to depend on the possibility of altering the state of the environment [Ho, Pa], so that at some iteration the termination condition might eventually hold.

There is at least one attempt to introduce iteration in Prolog without destructive assignment (see [CCP], for example). It involves a predicate **repeat**, whose purpose is to provide an endless backtracking point. On backtracking, all variables become again uninstantiated - so they can take new values, as needed - but then it is not possible to associate with them new values computed from the previous ones, since these are lost.

In the next section we recall a well-known result from the theory of computation: the transformation of recursive into iterative programs. Our objective is to find an applicative counterpart to the destructive assignment of conventional iteration.

3. ITERATION ORIGINATED FROM TAIL RECURSION

We start with a preliminary step: the transformation of a sub-class of the primitive recursive functions into tail-recursive functions [BW]. The simplest example is factorial, usually expressed by:

```
factorial(n) ≡ n = 0 -> 1 ; n * factorial(n-1)
```

This initial formulation is not tail recursive, since something remains to be done after the recursive call, namely the multiplication of its result by the current value of n. The execution of non-tail recursive functions requires a stack discipline to recover the values of parameters on finishing a recursive call. Keeping a stack can be avoided, thus enhancing efficiency, if an equivalent tail recursive expression is used:

```
factorial(n,m) ≡ n = 0 -> m ; factorial(n-1,n*m)
```

where the function is to be initially called to compute, say, the factorial of 5, as follows:

```
factorial(5,1)
```

By contrast, the example function below, to compute the greatest common divisor, is already in tail recursive form, and hence no transformation is required:

```
gcd(n,m) ≡ m = 0 -> n ; gcd(m,n mod m)
```

to be called by:

```
gcd(24,9)
```

Clever Prolog interpreters recognize and take advantage of tail recursion [Ca]. For procedural languages one is advised to further transform tail recursive into iterative programs [Bar2, BW]. Indeed, this transformation is illustrated by the two programs that we have shown to evaluate the greatest common divisor (the above function and the **while** loop at the beginning of section 2).

The simple point to note in comparing the two programs is that the gcd function contains no destructive assignment to variables. Instead, the variables are treated as **parameters** that, at each call, may take different values, usually derived from the values they had in the previous call. Initialization of variables takes the guise of the original call to the function. On termination of the evaluation no value remains associated with the parameters, as opposed to what happens to ordinary variables that are part of the environment in von Neumann languages. Needless to say, "temporary" variables are no longer needed to emulate parallel execution, although one still might have local variables in the body of a function, whenever convenient, without the problems attributed to global variables.

At this point, the reader might conclude that what we have done

was to find a motivation for a restricted use of destructive assignment. It would seem that some cases of destructive assignment are as justified as certain uses of go-to's. One could admit them inside while-like constructs in LISP or Prolog as a tolerable exception to the principles of applicative programming. Yet, we shall insist that iteration can be introduced without destructive assignment. Having found that destructive assignment is not needed if, inside recursive functions, variables are treated as parameters, what is still missing in our argument is a transformation that eliminates recursion without leading to the standard von Neumann iteration.

In the next section, we do this transformation in two stages. At the first stage, we obtain a non-recursive function and show what it means to evaluate it repeatedly. At the second stage, the in-line iterative construct is finally obtained.

4. ITERATION AS EXPONENTIATION

Iteration has been considered as a possible **functional form** to be incorporated into functional programming languages. In [Bac], the **while** form is defined as follows:

```
(while p f):x ≡
  p:x = T -> (while p f):(f:x);
  p:x = F -> x;
  ⊥
```

The same form is defined in terms of **functional exponentiation** in [F1], noting that the author uses postfix notation:

```
x:(while b do f) ≡
  if there exists n≥0, so that x:(f↑n & b) = F and
     for all 0≤m<n, x:(f↑m & b) = T
  -> x:f↑n;
  ⊥
```

Exponentiation means the composition of a function with itself an indicated number n of times. Alternatively, one can leave n unspecified and require that composition should proceed until a certain criterion is met [BL]. Fixed exponents resemble **for** iterators, whereas variable exponents correspond to **while** iterators.

For the case of variable exponents, consider a function **f** applied to a list of parameters X. Recall that, according to the usual conventions for exponentiation, we may regard its application as generating a sequence X0, X1, ..., where Xi = f↑i(X), so

```
X0 = X
Xi+1 = f(Xi)
```

Then, two termination criteria seem especially appropriate:

(a) f(Xi) = Xi - the fixed-point criterion
(b) f(Xi) = ⊥ - the failure criterion

In both (a) and (b) we take Xn as the result of applying f↑n to X, letting n be the value of the least i≥0 for which either (a) or (b) holds.

We can use function exponentiation directly, rather than as a device to introduce other concepts. With exponentiation we now give a third, not explicitly recursive, characterization of factorial. Notice that, to facilitate composition, the function is unary, and both its single argument and its result take the same list format:

```
factorial([n,m]) ≡
  n = 0 -> [n,m];
  #[n-1,n*m]
```

where #L denotes the list resulting from evaluating each member of L. This notation stresses the possibility of parallel evaluation.

To evaluate the function repeatedly, corresponds to raising it to an exponent, as in:

```
factorial↑i([5,1])
```

We are now in a position to perform the last step towards an in-line iterative construct. To this end, we note that the same non-recursive function definition above can be readily embedded in a larger program. One can regard this embedding as an application of the idea of "nameless" functions, introduced with the **lambda calculus** [La]. The exponentiation operation itself also becomes part of the construct.

```
 ............
[n1,m1] := i↑[n:5,m:1]--
   n = 0 -> [n,m] ;
   #[n-1,n*m]
 ............
```

The first line is the **heading** and what follows is the **body** of the iteration construct, the latter consisting of one simple or composite command. The syntax used above is not the important issue, being largely a matter of taste. Far more relevant is to recognize the various notions that participate in the construct:

(a) parameters: n and m
(b) initial values of the parameters: 5 and 1
(c) results: n1 and m1
(d) exponent: i

Parameters are initialized on entering the construct and may have different values at each iteration. Such values are usually computed from the previous ones. On exit, the final values of the parameters are associated with the results whereas the parameters themselves revert to the undefined condition they had prior to initialization. The exponent registers the current iteration, during execution; on exit, it indicates the number of iterations performed before a termination criterion is satisfied.

Other possible components are:

(e) exponent value, whenever it is known beforehand, as in i:v ↑ ..., where i is the exponent and v is the value
(f) local variables, whose value is undefined at the beginning of each iteration and on termination.

If two (or more) iterations are nested, the exponent and results of the inner iteration are regarded as local variables of the outer iteration. Accordingly, their values are available within the scope of the outer iteration, but become uninstantiated when the latter ends.

5. A PROTOTYPE IMPLEMENTATION

In order to demonstrate these ideas, we wrote a prototype implementation in Prolog. It works on predicates, as usual in Prolog, rather than on functions. We also produced an implementation to handle functions, as introduced in Prolog in an earlier report [Fu]; space considerations prevent its inclusion here.

To be run by the prototype, in predicate notation, the factorial example must be rewritten as indicated below. The conventions are compatible with those of the Edinburgh Prolog de facto standard [CCP]. The operators "=" and ":=" denote **unification**, in the case of the latter preceded by the evaluation of the right-hand side expression. When used in the iteration body, the result variables (N1 and M1 in the example) refer to the new values of the parameters (N and M, respectively, in the example).

```
 ............
[N1,M1] \\ I@[N:5,M:1]--
   case([N == 0 -> [N1,M1] = [N,M] ;
         [N1,M1] := #[N-1,N*M]]),
 ............
```

The implementation regards the operator "\\" as a meta-predicate, behaving as an extension of the Prolog interpreter for commands inside its scope. To handle parameters, local variables and exponents, a **copy** utility (which is built-in for some Prolog dialects) reproduces expressions with new variables substituted for the original ones, thereby making it possible to iterate with different values. Nesting is handled as expected, by meta-predicate **from**, thanks to the depth-first execution order inherent in Prolog. When the value of the exponent is not fixed, we allow bactracking into an iteration loop; this can be used, for instance, to find all positions where a given element occurs in a list.

The "\\" operator emulates iteration by tail recursion. A more practical implementation would produce a new interpreter that would directly recognize and handle the construct. In particular, it would process parameters and local variables by an efficient technique, which might even lead to overwriting the corresponding memory cells in case of an underlying von Neumann machine.

Appendix A contains additional examples. The code for the prototype is listed in appendix B.

6. CONCLUSION

We have shown that exponentiation provides a kind of iteration congenial to applicative languages. Exponentiation can be applied to (non-recursive) functions defined separately, as done in [Fu], or to nameless expressions embedded in larger programs.

The fundamental idea is that certain variables, that are initialized before entering an iteration construct and destructively modified inside it, should instead be treated as parameters.

The meta-predicate-based prototype simply demonstrates that the approach is feasible. For practical use, a syntax appropriate to each applicative language under consideration should be devised, as well as efficient implementation algorithms at the interpreter or compiler level.

REFERENCES

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

[Bar1] D. W. Barron et al - The main features of CPL - Computer Journal 6 (1963) 134-143.

[Bar2] D. W. Barron - Recursive techniques in programming - Mcdonald (1968).

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

[BW] F. L. Bauer and H. Wössmer - Algorithmic language and program development - Springer (1982).

[Ca] J. A. Campbell (ed.) - Implementations of Prolog - Ellis Horwood (1984).

[CCP] H. Coelho, J. C. Cotta and L. M. Pereira - How to solve it with Prolog - Laboratorio Nacional de Engenharia Civil, Lisboa (1982).

[FCT] A. L. Furtado, M. A. Casanova and L. Tucherman - A framework for design/redesign experts - in Expert Database Systems - L. Kerschberg (ed.) - Benjamin Cummings (1987) 423-438.

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

[Fu] A. L. Furtado - Towards functional programming in Prolog - SIGPLAN Notices 3 (1988) 43-51.

[Ho] C. A. R. Hoare - An axiomatic basis for computer programming - CACM 12, 10 (1969) 576-583.

[La] P. J. Landin - A lambda-calculus approach - in Advances in programming and non-numerical computatioon - L. Fox (ed.) - Pergamon Press (1966).

[Pa] F. G. Pagan - Formal specification of programming languages; a panoramic primer - Prentice-Hall (1981).

```
% *** greatest common divisor

:- [N1,M1] \\ I@[N:24,M:9]--
     case([M == 0 -> [N1,M1] = [N,M] ;
          [N1,M1] := #[M,N mod M] ]),
   write(N1), nl.

result written: 3

% *** Fibonacci numbers

:- [M1,A1,B1] \\ I@[M:8,A:1,B:0]--
     case([M == 1 -> [M1,A1,B1] = [M,A,B] ;
          [M1,A1,B1] := #[M-1,A+B,A]]).

result written: 21

% *** Cartesian product - nested iterations

:-  [Xtail,T1] \\ I:3@[X:[a,b,c],T:[]]--
      (X = [X1|Xtail],
       append(T,F1,T1) from
       [Ytail,F1]\\ J:2 @[Y:[p,q],F:[]]--
          (Y = [Y1|Ytail],
           append(X1,Y1,P),
           append(F,[P],F1)) ),
    write(T1), nl.

result written: [[a,p],[a,q],[b,p],[b,q],[c,p],[c,q]]

% *** standard deviation - iterations plus other commands

std(L,S) :-
  [T1,Xtail] \\ I@[T:0,X:L]--
     (X = [X1|Xtail],
      T1 := T + X1),
  Av := T1/I,
  [T2,Xrest] \\ J:I@[T:0,X:L]--
     (X = [X1|Xrest],
      T2 := T + (X1 - Av) ^ 2),
  S := sqrt(T2 / (I-1)).

:- std([12,6,7,3,15,10,18,5],X), write(X), nl.

result written: 5.20988072

% ***  finding occurrences of z - iteration and backtracking

:- [R] \\ I@[L:[z,b,z,c]]--
     (L = [z|_], L = R ; L = [_|R]),
   not(R == []),
   write(I), nl,
   fail.

results written: 0, 2, no
```

APPENDIX B - Listing of Prolog prototype

```
% **** special symbols

:- op(700,xfx,\\).
:- op(300,xfy,@).
:- op(100,xfy,:).
:- op(200,xfy,--).
:- op(800,xfy,from).
:- op(700,xfx,:=).
:- op(200,fy,#).

% *** iteration processor

iter(X,E) :-
  is_iter(E,I,P,A,C),
  p_iter(I,C,P,A,X).

X \\ E :- iter(X,E).

is_iter(E,I,LL,A,C) :-
  nonvar(E),
  E = I @ (L -- C),
  is_iter1(L,LL,A).

is_iter1([],[],[]).
is_iter1([V : A | P],
         [V|Q],[A|R]) :-
  is_iter1(P,Q,R).

p_iter(K,E,P,V,X) :-
  (nonvar(K), K = I:N, ! ;
   K = I),
  p_it1(I,N,E,P,V,X,O).

p_it1(I,N,E,P,B,X,C) :-
  copy([E,P,I,X],
       [E1,P1,I1,X1]),
  I1 is C + 1,
  P1 = B,
  (C == N, !, X = B , I = C;
   p_it2(Z,X1,E1),
   ((Z == 'fails';
     not(Z == 'fails'),
     var(N), B == Z),
     X = B, I = C;
    not(Z == 'fails'),
    not(Z == B),
    D is C + 1,
    p_it1(I,N,E,P,Z,X,D))).

p_it2(Z,X,E) :-
  (E,  Z = X;
    not(E), !,
    Z = 'fails').

from(X,Y) :-
  Y,
  X.

% *** utilities

copy(X,Y) :-
  string_term(S,X),
  string_term(S,Y).

X := E :-
  case([E = #E1 -> ev_1(X,E),
        is_list(E) -> X = E ;
        X is E]).

is_list([]).
is_list([_|_]).

ev_1([],#[]).
ev_1([X|R],#[Y|S]) :-
  X is Y,
  ev_1(R,#S).

append(X,Y,Z) :-
  case([is_list(X) -> X1 = X ;
        X1 = [X]]),
  case([is_list(Y) -> Y1 = Y ;
        Y1 = [Y]]),
  append1(X1,Y1,Z).

append1([],X,X).
append1([A|R],X,[A|Y]) :-
  append1(R,X,Y).
```

Obs.: the **copy** utility may be affected by an inadequate maximum size fixed for the character string argument of **string_term.**