Declarative Loops and List Comprehensions for Prolog

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center
Last updated: January 25, 2010.

1 Introduction

Prolog relies on recursion to describe loops. This has basically remained the same since Prolog's inception 35 years ago. Many other languages provide powerful loop constructs. For example, the `foreach` statement in C# and the enhanced `for` statement in Java are very powerful for iterating over collections. Functional languages provide higher-order functions and list comprehensions for iterating over and creating collections. The lack of powerful loop constructs has arguably made Prolog less acceptable to beginners and less productive to experienced programmers because it is often tedious to define small auxiliary recursive predicates for loops. The emergence of constraint programming constructs such as CLP(FD) has further revealed this weakness of Prolog as a host language. ECLiPSe [1] provides logical loop constructs, but there are too many derivative iterators and the semantics are not simple (e.g., an iterator can be used to iterate through a list that is yet to be constructed).

In this note, we adapt to Prolog `foreach` for iterating over collections and list comprehensions for constructing lists. The `foreach` construct has a very simple syntax and semantics. For example, `foreach(A in [a,b], I in 1..2, write((A,I)))` outputs four tuples `(a,1)`, `(a,2)`, `(b,1)`, and `(b,2)`. Syntactically, `foreach` is a variable-length call whose last argument specifies a goal to be executed for each combination of values in a sequence of collections. A `foreach` call may also give a list of variables that are local to each iteration and a list of accumulators that can be used to accumulate values from each iteration. With accumulators, we can use `foreach` to describe recurrences for computing aggregates. Recurrences have to be read procedurally and thus do not fit well with Prolog. For this reason, we borrow list comprehensions from functional languages. A list comprehension is a list whose first element has the functor `':'/2`. A list of this form is interpreted as a list comprehension in calls to `'@='/2` and some other contexts. For example, the query `X@=[(A,I) : A in [a,b], I in 1..2]` binds `X` to the list `[(a,1),(a,2),(b,1),(b,2)]`. A list comprehension is treated as a `foreach` call with an accumulator in the implementation.

2 The Base `foreach`

The `foreach` construct is a variable-length call with the name `foreach`. The base form looks like:

`foreach(`E_1 `in` D_1`, ..., `E_n `in` D_n`, `$LocalVars$`,`$Goal$`)`

where E_i is normally a variable but can be any term, and D_i a list or a range of integers $l..u$ (inclusive), $LocalVars$, which is optional, is a list of variables in $Goal$ that are local to each iteration, and $Goal$ is a callable term. All variables in E_i's are local variables. The `foreach` call means that for each combination of values $E_1 \in D_1, \ldots, E_n \in D_n$, the instance $Goal$ is executed after local variables are renamed. The call fails if any of the instances fails. Any variable that occurs in $Goal$ but is not in any E_i or $LocalVars$ is shared by all iterations.

Examples

```
?-foreach(I in [1,2,3],write(I)).
123

?-foreach(L in [[1,2],[3,4]], (foreach(I in L, write(I)),nl)).
12
34

?-functor(A,t,10),foreach(I in 1..10,arg(I,A,I)).
A = t(1,2,3,4,5,6,7,8,9,10)

?-foreach((A,I) in [(a,1),(b,2)],writeln(A=I)).
a=1
b=2
```

The power of `foreach` is more clearly revealed when it is used with arrays. The following predicate creates an `N`×`N` array, initializes its elements to integers from 1 to `N`×`N`, and then prints it out.

```
go(N):-
   new_array(A,[N,N]),
   foreach(I in 1..N,J in 1..N,A[I,J] is (I-1)*N+J),
   foreach(I in 1..N,
           (foreach(J in 1..N,
                    [E],(E @= A[I,J], format("~4d ",[E]))),nl)).
```

In the last line, `E` is declared as a local variable. In B-Prolog, a term like A[I,J] is interpreted as an array access in arithmetic built-ins, calls to '@='/2, and constraints, but as the term `A^[I,J]` in any other context. That is why we can use `A[I,J] is (I-1)*N+J` to bind an array element but not `write(A[I,J])` to print an element.

3 Simultaneous `foreach`

The base `foreach` cannot be used to easily iterate over multiple collections simultaneously. For example, given two lists, it is not easy to use the base `foreach` to create an association list from the two. To facilitate iteration over multiple collections simultaneously, we allow an iterator to take the form

$$(X_1, X_2, \ldots, X_n) \texttt{ in } (D_1, D_2, \ldots, D_n)$$

which means that for each $X_1 \in D_1$, an element X_i is picked from D_i (i$\geq$2) simultaneously. For example,

```
?-foreach((X,Y) in ([a,b],1..2),writeln(X=Y)).
a=1
b=2
```

4 `foreach` with Accumulators

The base `foreach` is not suitable for computing aggregates. We extend it to allow accumulators. The extended `foreach` takes the form:

$$\texttt{foreach(}E_1 \texttt{ in } D_1\texttt{, ..., } E_n \texttt{ in } D_n\texttt{, } LocalVars\texttt{,}Accs\texttt{,}Goal\texttt{)}$$

or

$$\texttt{foreach(}E_1 \texttt{ in } D_1\texttt{, ..., } E_n \texttt{ in } D_n\texttt{, } Accs\texttt{,}LocalVars\texttt{,}Goal\texttt{)}$$

where *Accs* is an accumulator or a list of accumulators. The ordering of $LocalVars$ and *Accs* is not important since the types are checked at runtime.

One form of an accumulator is `ac`$(AC, Init)$, where AC is a variable and $Init$ is the initial value for the accumulator before the loop starts. In *Goal*, recurrences can be used to specify how the value of the accumulator in the previous iteration, denoted as AC`^0`, is related to the value of the accumulator in the current iteration, denoted as AC`^1`. Let's use $Goal(AC_i, AC_{i+1})$ to denote an instance of *Goal* in which AC`^0` is replaced with a new variable AC_i, AC`^1` is replaced with another new variable AC_{i+1}, and all local variables are renamed. Assume that the loop stops after n iterations. Then this `foreach` means the following sequence of goals:

$AC_0 = Init$,
$Goal(AC_0, AC_1)$,
$Goal(AC_1, AC_2)$,
...,
$Goal(AC_{n-1}, AC_n)$,
$AC = AC_n$

Examples

```
?-foreach(I in [1,2,3],ac(S,0),S^1 is S^0+I).
S = 6

?-foreach(I in [1,2,3],ac(R,[]),R^1=[I|R^0]).
R = [3,2,1]

?-foreach(A in [a,b], I in 1..2, ac(L,[]), L^1=[(A,I)|L^0]).
L = [(b,2),(b,1),(a,2),(a,1)]

?-foreach((A,I) in ([a,b],1..2), ac(L,[]), L^1=[(A,I)|L^0]).
L = [(b,2),(a,1)]
```

The following predicate takes a two-dimensional array, and returns its minimum and maximum elements:

```
array_min_max(A,Min,Max):-
    A11 is A[1,1],
    foreach(I in 1..A^length,
            J in 1..A[1]^length,
            [ac(Min,A11),ac(Max,A11)],
            ((A[I,J]<Min^0->Min^1 is A[I,J];Min^1=Min^0),
             (A[I,J]>Max^0->Max^1 is A[I,J];Max^1=Max^0))).
```

A two-dimensional array is represented as an array of one-dimensional arrays. The notation `A^length` means the size of the first dimension.

Another form of an accumulator is $ac1(AC, Fin)$, where Fin is the value AC_n takes on after the last iteration. A `foreach` call with this form of accumulator means the following sequence of goals:

$$
\begin{array}{l}
AC_0 = FreeVar, \\
Goal(AC_0, AC_1), \\
Goal(AC_1, AC_2), \\
\ldots, \\
Goal(AC_{n-1}, AC_n), \\
AC_n = Fin, \\
AC = FreeVar
\end{array}
$$

We begin with a free variable $FreeVar$ for the accumulator. After the iteration steps, AC_n takes on the value Fin and the accumulator variable AC is bound to $FreeVar$. This form of an accumulator is useful for incrementally constructing a list by instantiating the variable tail of the list.

Examples

```
?-foreach(I in [1,2,3], ac1(R,[]), R^0=[I|R^1]).
R = [1,2,3]

?-foreach(A in [a,b], ac1(L,Tail), L^0=[A|L^1]), Tail=[c,d].
L = [a,b,c,d]

?-foreach((A,I) in ([a,b],1..2), ac1(L,[]), L^0=[(A,I)|L^1]).
L = [(a,1),(b,2)]
```

5 List Comprehensions

A list comprehension is a construct for building lists in a declarative way. List comprehensions are very common in functional languages such as Haskell, Ocaml, and F#. We propose to introduce this construct into Prolog.

A list comprehension takes the form:

$$\texttt{[}T\ \texttt{:}\ E_1\ \texttt{in}\ D_1\texttt{,}\ \ldots\texttt{,}\ E_n\ \texttt{in}\ D_n\texttt{,}\ LocalVars\texttt{,}Goal\texttt{]}$$

where $LocalVars$ (optional) specifies a list of local variables, $Goal$ (optional) must be a callable term. The construct means that for each combination of values $E_1 \in D_1$, ..., $E_n \in D_n$, if the instance of $Goal$, after the local variables being renamed, is true, then T is added into the list.

Note that, syntactically, the first element of a list comprehension takes the special form of T:(E in D). A list of this form is interpreted as a list comprehension in calls to `'@='/2` and constraints in B-Prolog.

A list comprehension is treated as a `foreach` call with an accumulator. For example, the query `L@=[(A,I) : A in [a,b], I in 1..2]` is the same as

```
foreach(A in [a,b], I in 1..2, ac1(L,[]),L^0=[(A,I)|L^1])
```

.

Examples

```
?-L @=[X : X in 1..5].
L = [1,2,3,4,5]

?- L @= [1 : X in 1..5].
L = [1,1,1,1,1]

?- L @= [Y : X in 1..5].
L = [Y,Y,Y,Y,Y]
```

```
?- L @= [Y : X in 1..5,[Y]]. % Y is local
L = [_598,_5e8,_638,_688,_6d8]

?- L @= [Y : X in [1,2,3], [Y], Y is -X].
L = [-1,-2,-3]

?-L @=[(A,I): A in [a,b], I in 1..2].
L = [(a,1),(a,2),(b,1),(b,2)]

?-L @=[(A,I): (A,I) in ([a,b],1..2)].
L = [(a,1),(b,2)]
```

6 Application Examples

6.1 Print staircases

```
/* Draw a staircase of a given steps. Example query:

?-go(4).
            +---+
            |   |
        +---+---+
        |   |   |
    +---+---+---+
    |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
*/
go(N):-
    foreach(I in 1..N,
    (foreach(J in 1..N-I,  write('    ')),
     foreach(J in 1..I,  write('+---')),
     writeln('+'),
     foreach(J in 1..N-I,  write('    ')),
     foreach(J in 1..I,  write('|   ')),
     writeln('|'))),
    foreach(I in 1..N, write('+---')), writeln('+').
```

6.2 Quicksort

```
qsort([],[]).
qsort([H|T],S):-
```

```
    L1 @= [X : X in T, X<H],
    L2 @= [X : X in T, X>=H],
    qsort(L1,S1),
    qsort(L2,S2),
    append(S1,[H|S2],S).
```

6.3 Generate permutations

```
perms([],[[]]).
perms([X|Xs],Ps):-
    perms(Xs,Ps1),
    Ps @= [P : P1 in Ps1, I in 0..Xs^length,[P],insert(X,I,P1,P)].

insert(X,0,L,[X|L]).
insert(X,I,[Y|L1],[Y|L]):-
    I>0,
    I1 is I-1,
    insert(X,I1,L1,L).
```

6.4 N-Queens problem

```
queens(N):-
    length(Qs,N),
    Qs :: 1..N,
    foreach(I in 1..N-1, J in I+1..N,
                (Qs[I] #\= Qs[J],
                 abs(Qs[I]-Qs[J]) #\= J-I)),
    labeling([ff],Qs),
    writeln(Qs).
```

In B-Prolog (version 7.4 and up), the array subscript notation X[I1,...,In] can be used to access arguments of structures and elements of lists. An array is just a structure whose functor is '[]', and a multi-dimensional array is a structure of structures. In this example, `Qs` is a list and `Qs[I]` means the `I`th element of the list.

6.5 N-Queens problem as a SAT problem

```
bool_queens(N):-
    new_array(Qs,[N,N]),
    Vars @= [Qs[I,J] : I in 1..N, J in 1..N],
    Vars :: 0..1,
    foreach(I in 1..N,
            sum([Qs[I,J] : J in 1..N]) #= 1),
```

```
    foreach(J in 1..N,
            sum([Qs[I,J] : I in 1..N]) #= 1),
    foreach(K in 1-N..N-1,
            sum([Qs[I,J] : I in 1..N, J in 1..N, I-J=:=K]) #=< 1),
    foreach(K in 2..2*N,
            sum([Qs[I,J] : I in 1..N, J in 1..N, I+J=:=K]) #=< 1),
    labeling(Vars),
    foreach(I in 1..N,[Row],
            (Row @= [Qs[I,J] : J in 1..N], writeln(Row))).
```

6.6 Magic squares

```
go(N):-
    new_array(Board,[N,N]),
    NN is N*N,
    Vars @= [Board[I,J] : I in 1..N, J in 1..N],
    Vars :: 1..NN,
    Sum is NN*(NN+1)//(2*N),
    foreach(I in 1..N,sum([Board[I,J] : J in 1..N]) #= Sum),
    foreach(J in 1..N,sum([Board[I,J] : I in 1..N]) #= Sum),
    sum([Board[I,I] : I in 1..N]) #= Sum,
    sum([Board[I,N-I+1] : I in 1..N]) #= Sum,
    all_different(Vars),
    labeling([ffc],Vars),
    foreach(I in 1..N,
    (foreach(J in 1..N, [Bij],
                    (Bij @= Board[I,J], format("~4d ",[Bij]))),nl)).
```

7 A Comparison with ECLiPSe's Loop Constructs

Besides recursion, Prolog provides other constructs, such as failure-driven control constructs and higher-order predicates, for describing loops. These facilities, however, do not fit well with CLP(FD) because no variables can be retained after looping. Our proposed loop constructs were inspired by ECLiPSe's loop constructs, but they are different both syntactically and semantically.

Syntactically, ECLiPSe provides a built-in, called do/2, in which several types of iterators (`fromto/4`, `foreach/2`, `for/3`, `for/4`, `foreacharg/2`, `count/3`, and `param/n`) can be specified. In contrast, our proposed `foreach` is a variable-length call in which only one type of iterator `E in D` is needed. An iterator specifies a pattern for an element and a collection which can be a list or a range of integers. An iterator with a tuple of collections specifies a simultaneous loop. Accumulators can be given to describe recurrences for computing aggregates and the list comprehension notation is just a foreach call that takes one accumulator.

Semantically, ECLiPSe's iterators are unification-based while our iterators are matching-based. In ECLiPSe, the collection of an iterator can be changed by unification even if the goal of the loop does not change anything. For example,

```
?-foreach(f(a,X),[f(a,b),f(Y,Z)]) do write(X).
```

displays `b` and `Z`, and as a side effect, binds `Y` to `a` after the loop. If there is an element in the list that does not unify with the pattern, then the whole loop fails. For example,

```
?-foreach(f(a,X),[c,f(a,b),f(Y,Z)]) do write(X).
```

fails. In contrast, our iterators never change a collection unless the goal of the loop has that effect. For example,

```
?-foreach(f(a,X) in [c,f(a,b),f(Y,Z)],write(X)).
```

displays `b`. The elements `c` and `f(Y,Z)` are skipped because they do not match the pattern `f(a,X)`.

In ECLiPSe, variables are assumed to be local to each iteration unless they are declared global in a `param` iterator. In contrast, in our proposed construct, variables are assumed to be global to all the iterations unless they are declared local or occur in the patterns of the iterators. From the programmer's perspective, the necessity of declaring variables is a burden in both approaches. We believe, however, that there are normally fewer local variables than global ones, and hence the global-by-default approach imposes less a burden on the programmer. Sometimes, people may use anonymous variables '_' in looping goals and wrongly believe that they are local. This is a weakness of the global-by-default approach. Nevertheless, with warnings from the reader, this problem is alleviated.

From the implementation perspective, the difference between global-by-default and local-by-default is minor when loops are compiled. When loops are interpreted, however, the advantage of global-by-default is obvious because only local variables in the looping goal, not the entire goal itself, needs to be copied.

Our constructs are more convenient than ECLiPSe's for describing nested loops. Figure 1 compares the encodings of the N-queens problem in B-Prolog and ECLiPSe . In B-Prolog, the operator `::` does not work on arrays, and so the list comprehension `Vars @= [Q[I] : I in 1..N]` is used to fetch the variables from the array. The loop variables `I` and `J` are local by default. In comparison, in the ECLiPSe's encoding, the variables `Qs` and `N` must be declared global, and in the inner loop even the loop variable `I` of the outer loop has to be declared global.

In ECLiPSe, it's possible to use iterators to construct multiple lists of the same length. For example,

```
?-foreach(X,[1,2,3]),foreach(Y,Ys),foreach(Z,Zs) do
    Y is -X, Z is abs(X).
```

B-Prolog

```
queens(N):-
  new_array(Qs,[N]),
  Vars @= [Q[I] : I in 1..N],
  Vars :: 1..N,
  foreach(I in 1..N-1, J in I+1..N,
    (Qs[I] #\= Qs[J],
    abs(Qs[I]-Qs[J]) #\= J-I)),
  labeling([ff],Qs)
```

ECLiPSe

```
queens(N):-
  dim(Qs,[N]),
  Qs :: 1..N,
  (for(I,1,N-1), param(Qs,N) do
    (for(J,I+1,N), param(Qs,I) do
      Qs[I] #\= Qs[J],
      abs(Qs[I]-Qs[J]) #\= J-I
    )
  ),
  labeling([ff],Qs).
```

Fig. 1. Encodings of the N-queens problem in B-Prolog and ECLiPSe.

builds two lists `Ys` and `Zs` from `[1,2,3]`. In general, there are more cases where we need to iterate over multiple collections than where we need to build multiple collections. For the later cases, we can either use multiple accumulators or `maplist/n`, which is as convenient as ECLiPSe's iterators.

Acknowledgement

Both the design and the implementation in B-Prolog benefited greatly from discussions in the comp.lang.prolog news group. Thanks to the following people for their participation in the discussions: Bart Demoen, Ulrich Neumerkel, Paulo Moura, Joachim Schimpf, Kish Shen, Markus Triska and Jan Wielemaker. Special thanks to Joachim Schimpf for his explanations of the loop constructs in ECLiPSe. The operator ':' for list comprehension was chosen from a list of suggested operators by Ulrich Neumerkel.

References

1. Joachim Schimpf. Logical loops. In *ICLP*, pages 224–238, 2002.