A General-Purpose Conjunctive Iterative Control Structure for Prolog

Yasusi Kanada[1]

Abstract

A loop-like control structure without using backtracking, or conjunctive iteration, is expressed using recursion in Prolog. However, recursion is too powerful to express an iteration, which needs more restrictive syntax and semantics. This paper presents a general-purpose iteration predicate **do**. Predicate **do** enables a programmer to write most iterations, such as arithmetical iterations, **append**, **member**, **mapcar** or **reduce**, and so on, more easily and in more readable way, in combination with the extended λ term, which is a concept similar to the λ expression in Lisp. Unification and logical variables in Prolog enables some extensive usage of the control structure compared with those of other programming languages, such as Lisp.

1. Introduction

One of the problems encountered when using Prolog in a large scale application is the lack of structures, i.e., the lack of modules and control structures, limited data structures, and so on. The only way to write conjunctive iteration, or a loop-like control structure without using backtracking, in Prolog is to write it using recursion. However, recursion is too powerful to write simple iterations. Recursion is so powerful that it may lead programmers to an unreadable and undebuggable programming style. Restricted iterative control structure, or predicate, is more suited for writing simple iterations than recursion.

Some disjunctive iterative control structures, or in other words, iterative backtracking control structures, can be defined using predicate **repeat** in Prolog or using other predicates defined in several papers. Predicate **repeat** is used for making a disjunctive iterative control structure, though it is not a *structured* one. Dodson and Rector [Dod 83] define several structured disjunctive control structures for the purpose of structured use of cut, **fail** and **repeat**. Munakata [Mun 86] also shows some structured control structures. Some conjunctive iteration predicates, or in other words, tail recursion control structure, such as **mapcar**, also appeared in some papers or programs, e.g., Kondoh and Chikayama [Kon 87]. However, these conjunctive iteration predicates can be used only for limited purposes.

This paper explains a general-purpose conjunctive iterative control predicate **do**, which enables a programmer to write most conjunctive iterative control structures, such as arithmetical iterations, **append**, **member**, **mapcar**, and so on, more easily and in more readable way, in combination with the extended λ term, which is a concept similar to λ expression in Lisp. Section 2 shows the definitions of λ term and predicate **do**. Section 3 shows a usage of **do** in arithmetical iterations, in iterative list processing and in mapping. In Section 4, we dis-

[1] The author's address is as follows: Center for Machine Translation, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA.

cuss the advantages and disadvantages of **do** compared to recursion, and we also discuss the efficiency of **do**.

2. Definitions of λ term and do

2.1 λ term

A λ term is an unnamed clause or predicate. The syntax of a λ term is as follows:

term **** *term*

The left-hand side of **** is usually a list of terms: **[***term***,** *term***,** …**,** *term***]**; it is called the *lambda list.* The right-hand side is called the *lambda body*. The semantics of λ term is close to λ expression in Lisp:

(lambda (*var***,***var***,** … **,***var***)** *expression***)**

The first *term* in λ term corresponds to the variable list in λ expression, and the second *term* corresponds to the *expression*.

However, the form of each *term* in the lambda list is not limited to a variable, but it can be a list or other structured term and "λ binding" is done by unification. A λ term is defined in λ Prolog [Nad 88], but the major difference between ours and λ Prolog's is in this point.

For example, instead of defining predicate **add** as

```
add(X,Y,Z) :- Z is X + Y.
```

and using it as **add(1,2,R)**, we can use a λ term **[X,Y,Z]\(Z is X + Y)**. Instead of using **add(1,2,R)**, we can *apply* the predicate to a list by the following term:

```
apply([X,Y,Z]\(Z is X + Y), [1,2,R]).
```

Though there is no strict reason to use a list for binding the lambda variables **X**, **Y**, **Z** and the arguments **1**, **2**, **R**, it is a good convention to do so. (Note that the following term has the same meaning as the above one: **apply(f(X,Y,Z)\(Z is X + Y), f(1,2,R)**.) The unification-based binding is so powerful that a lot of predicate can be written with an empty lambda body, such as **[…]\true**. Examples are shown later.

If predicate **apply** is used for applying a globally defined predicate *p*, such as **add**, the result caused by **apply(***p***,** *l***)** is equivalent to that by **(X=..[***p***|** *l***], call(X))** where **X** is a new variable. A simple but inefficient implementation of **apply** in Prolog is shown in Appendix.

An *or-composition* of two λ terms is defined as follows: if t_1 and t_2 are λ terms, or-composition of t_1 and t_2 is written t_1**;** t_2. The meaning of term **apply(**t_1**;** t_2**,** l**)** is equivalent to that of term **(apply(**t_1**,** l**); apply(**t_2**,** l**))**. An *and-composition* can be defined in the same way, but it is omitted here.

For example, the definition of the recursive append predicate is shown below.

```
def append =
   ([[],X,X]\true;  [[H|T1],Y,[H|T2]]\append(T1,Y,T2))).
```

2.2 Predicate do

We can define predicate **do** with two or more parameters: **do(P,X1-X1e,** … **,Xn-Xne)**. The first parameter, **P**, is the predicate to be applied repeatedly. It may be called the "loop body." Other parameters are the data to be applied, and they are *difference data* (*d-data*). D-data are expressed by structures with two arguments, **Xi-Xie** which is semantically the same as **'-'(Xi,Xie)**. A d-data is, in a sense, an extension of a *d-list* (*difference list*). If it is used as a parameter of a predicate, it can also be regarded as a Prolog counterpart of a variable or variable parameter in procedural languages. For example, in **do(P,-[X|Xe]-Xe)**, the second parameter can be regarded as a single-element d-list of **X**, and it can also be informally regarded as a variable parameter whose input value is **[X|Xe]** and output value is **Xe**. Concrete examples are shown in the following section.

A *part* of each d-data is used as a parameter in the predicate application. The informal meaning of a *part* is explained below. If **Xi-Xie** is a d-data, then **Xi-Xi1**, **Xi1-Xi2**, …, and **Xin-Xie** are interpreted as *parts* of the d-data. If **Xi-Xie** is a d-list, a part of the d-data (d-list) means a sublist of the d-list. For example, d-list **[b,c|X]-X** is a *part* of d-list **[a,b,c,d|Xe]-Xe**.

Predicate **do** with one parameter can be defined as follows:

```
do(P,A1-A1).
do(P,A1-A1e) :- apply(P,[A1-A1a]), do(P,A1a-A1e).
```

And predicate **do** with two parameters can be defined as follows:

```
do(P,A1-A1,A2-A2).
do(P,A1-A1e,A2-A2e) :-
   apply(P,[A1-A1a,A2-A2a]), do(P,A1a-A1e,A2a-A2e).
```

Here, **A1-A1a** and **A1a-A1e** are *parts* of d-data **A1-A1e**. **A2-A2a** and **A2a-A2e** are *parts* of d-data **A2-A2e**.

Predicates **do** with more than two parameters can also be defined in the same way.

3. Usage of do

Three kinds of usage are shown below: arithmetical iteration, iterative list processing, and mappings.

3.1 Arithmetical iteration

The first example of arithmetical iteration is the factorial function. Predicate call **fac1(X,Y)** inputs the value of **X** and outputs the value of **X**! as **Y**.

```
fac1(X,Y) :- do(fac1a, 0-X, 1-Y), !.

fac1a(I-I1,J-J1) :- I1 is I + 1, J1 is J * I1.
```

The second definition can be rewritten using a λ term.

```
fac1(X,Y) :-
   do([I-I1,J-J1]\(I1 is I + 1, J1 is J * I1), 0-X, 1-Y), !.
```

The second parameter of **fac1** can be regarded as the values of the loop control variable. Its initial value is **0**, and its final value is **X**, which is defined in the execution of **fac1a** or the λ term. The "control variable" is incremented by 1 in each iteration step.

In each iteration step, the value of the "control variable" is inputted to **fac1a** or the λ term as **I**, and its new value is outputted as **I1**. The second parameter is handled in the same way. **J** is inputted and the new value of **J** is defined as **J*I** where **I** is the old value. D-data, **I-I1** and **J-J1**, can be informally regarded as variable parameters here. The initial values of **I** and **J** are **0** and **1** respectively, and the above step is repeated until the value of **I** becomes equal (unifiable) to **X**. The result **Y** is the value of **J** at the end of the repetition.

In fact, recursive factorial predicate can be defined more simply.

```
fac2(0,1) :- !.
fac2(I,J) :- I1 is I - 1, fac2(I1,J1), J is J1 * I.
```

However, **fac1** is more efficient than **fac2** if it is compiled by an optimizing compiler, because the recursion in **fac1** is a tail recursion while the recursion in **fac2** is not. A tail-recursive factorial predicate can be defined in conventional way as follow:

```
fac3(X,Y) :- fac3(0,X,1,Y).

fac3(I,I,J,J) :- !.
fac3(I,Ie,J,Je) :- I1 is I + 1, J1 is J * I1,
fac3(I1,Ie,J1,Je).
```

fac3 is as efficient as **fac1**, but **fac1** seems to be simpler and more structured than **fac3** and at least it is less redundant because there is no repetition of the predicate name **fac3**. More detailed comparison is given in Section 4.

The second example is Newton's method. Predicate **newton** calculates the solution of the equation $f(x) = \sqrt{2}\,x^3 + 2x^2 - 1 = 0$.

```
newton(X) :- do(newton1, 1-X, 999-Eps), Eps<1e-5.

newton1(X-X1,_-Eps) :-
   f(X,Fx,Dfdx), D is Fx / Dfdx, X1 is X - D, abs(D,Eps).

f(X,Fx,Dfdx) :-
   Fx is sqrt(2)*X^3 + 2*X^2 - 1, Dfdx is sqrt(18)*X^2 + 4*X.

abs(X,Y) :- X >= 0 -> Y is X; Y is -X.
```

Here, predicate **f** computes $f(x)$ and $f'(x)$, and **abs(X,Y)** results **Y** = $|$**X**$|$. Constant **999** in predicate **newton** is an arbitrary value which is greater than **1e-5**. The loop termination condition, **Eps<1e-5**, is tested outside of the **do** construct because a comparison of numbers cannot be described in a call of predicate **do**. A loop termination condition can be described in the actual parameters of **do** only when it can be tested by unification as far as using the syntax and the semantics of DEC-10 Prolog.

3.2 Iterative list processing

Predicate **append** is defined using **do** as follows.

```
append(X,Y,Z) :- do([[H|T1]-T1,[H|T2]-T2]\true, X-[], Z-Y).
```

The first parameter of **do** is a λ term predicate that unifies each element of a list with each element of another list, or, in other words, that copies a single-element d-list. The second parameter of **do** can be regarded as the control variable when the first and second parameters of **append** are used as input parameters and the third as an output parameter. **X** is the initial value and **[]** is the final value.

The λ term in the **do** construct unifies an element of d-list **X-[]** with an element of d-list **Z-Y** in each iteration step, until the value of the "control variable" becomes **[]**. Because the last half of d-list **Z-Y** is **Y**, **Z** becomes the concatenation of the copied list and **Y**, the second argument of **append**, after executing the loop.

The above predicate **append** can be used in reverse direction. For example, if the second and third parameters are used as input and the first as output, **append** finds a list, **X**, which satisfies the relation **append(X,Y,Z)**. In this case, the third parameter of **do** works as the "control variable." Its initial value is **Z** and its final value is **Y**. This exchangeability of "control variables" is the most remarkable feature of **do**, because control variables are syntactically fixed in conventional iterative control structures of procedural languages. The above **append** can also be used for nondeterministically decomposing a list into two lists the same as the recursively-defined **append**.

The definition of predicate append is slightly complicated in its syntax because two d-lists are appeared. If a syntactic sugaring for d-lists is introduced, it becomes simpler. We abbreviate d-list **[**x_1**,** x_2**,** …**,** x_n**|** y**]-**y to **<**x_1**,** x_2**,** …**,** x_n**>**. Then the above definition becomes

```
append(X,Y,Z) :- do([<H>,<H>]\true, X-[], Z-Y).
```

This abbreviation is possible because variables **T1** ans **T2** are not used outside of the d-lists.

Other iterative list processing predicates are also defined using **do**. The following examples are some of them.

```
member(X,Y) :- do([<_>\true, Y-[X|_]).

reverse(X,Y) :- do([<H>,T1-[H|T1]]\true, X-[], []-Y).

length(X,Y) :- do([<_>,L-L1]\(L1 is L + 1), X-[], 0-Y).
```

Predicate **member** tests whether the first parameter **X** is a member of the second parameter, list **Y**. Predicate **reverse** is an optimized version of predicate that reverses a list. An "optimized version" means it is not a so-called *naive reverse* which takes $O(n^2)$ time for list reversal. Predicate **length** that counts the number of elements in the list which is passed as the first parameter, **X**. It can also be used in reverse direction as **length(X,5)**, which results in a five element list, all the elements of which are logical variables.

3.3 Mappings

Two examples of mappings are shown below, i.e., **mapcar** and **mapcan**.

The first example is predicate **mapcar**, whose function is similar to the function of the same name in Lisp, is defined as follows:

```
mapcar(P,X,Y) :-
   do([<H1>,<H2>]\apply(P,[H1,H2]), X-[], Y-[]).
```

The first parameter of **mapcar** is a λ term predicate, and the second and third parameters are lists. The predicate is applied to each pair of elements of the lists.

The second parameter of **mapcar** can be used as input and the third as output. For example, **mapcar(length,[[a],[b,c,d],[e,f]],Y)** results in **Y=[1,3,2]**. The second parameter works as the "control variable" in this case. Predicate **mapcar** is symmetric on the second and third parameters, so the second parameter can be used as output, and the third as input. The third parameter works as the "control variable" in this case.

Predicates **mapcar** with two arguments (**mapcar/2**) or more than three arguments can also be defined in the same way. The function of the former is close to that of function **mapc** in Lisp because **mapcar/2** does not have an argument for returning a result.

The second example is predicate **mapcan**. In Lisp, **mapcan** is a convenient function which may generates zero or more than one element of the resulting list from each element of the input list. For example, function **double**, which doubles each element of the input list, is defined as follows:

```
(defun double (x) (mapcan #'(lambda (h) `(,h ,h)) x))
```

For example, expression **(double '(a b c))** is evaluated to be **'(a a b b c c)**.

Function **mapcan** destructively concatenates the list that the function given as the first parameter of **mapcan** returns. So it is impossible to rewrite this program into Prolog directly. A d-list can be used to simulate the destructive concatenation. Then, the Prolog version of **mapcan** and **double** can be defined as follows:

```
mapcan(P,X,Y) :- do([<H>,DL]\apply(P,[H,DL]), X-[], Y-[]).

double(X,Y) :- mapcan([E,<E,E>]\true, X, Y).
```

The first parameter of **mapcan** is a λ term predicate; its first parameter is an element of list **X** and its second parameter, **DL**, is a d-list. Predicate **mapcan** outputs the concatenation of these d-lists as list **Y**.

However, it is simpler to define **double** directly using **do**:

```
double(X,Y) :- do([<H>,<H,H>]\true, X-[], Y-[]).
```

For example, expression **double([a,b,c],Y)** results in **Y=[a,a,b,b,c,c]**. The above example shows that predicate **do** can be used for the programs for which function **mapcan** should be used if it is written in Lisp.

4. Discussion

The power of iterations using do is discussed in this section. The good and bad points of the description of loop termination conditions using **do**, and the efficiency of iterations using **do** are also discussed.

4.1 Extending the power of iterations

Some repetitions which cannot be written using iterations in pure Lisp or other languages can be written using do in *pure* way. This means, in a sense, predicate **do** extends the power of iterations. We will compare the two predicates with the counterparts of Lisp.

The iterative version of predicate **append** shown in Section 3.2 does not use any destructive operations. However, the function of predicate **append** cannot be implemented by a loop or a tail recursion in Lisp without using destructive operations such as **rplacd**. The normal definition of **append** in Lisp is as follows:

```
(defun append (x y)
  (if (null x)
      y
      (cons (car x) (append (cdr x) y))))
```

A list cell is to be created by the function **cons** after a recursion of **append**, so it is not a tail recursion and then it cannot be transformed to an iteration. If **append** is rewritten so that the list cell is created before the recursion, it can be a tail recursion, but the tail of the list cell must be rewritten by a non-pure destructive function, **rplacd**. The logical variable in Prolog (the third argument in this case) enable the iterative or tail-recursive **append** procedure.

Predicates **mapcan** shown in Section 3.3 are not a destructive operation because Prolog does not allow a destructive assignment. It calls a predicate repeatedly and concatenates the resulted d-lists into a normal list or a d-list by unifying logical variables. For example, when **double([a,b,c],Y)** shown in Section 3.3 is executed using the first version of **double** which calls mapcan, predicate **[E,<E,E>]\true** is called three times with the elements **a**, **b** and **c** respectively, and it returns **[a,a|T1]-T1**, **[b,b|T2]-T2** and **[c,c|T3]-T3** as the second arguments, where **T1**, **T2** and **T3** are arbitrary new logical variables. These d-

lists are concatenated by unifying **T1**, **T2** and **T3**, and **mapcan** and **double** returns the list **[a,a,b,b,c,c]**.

On the contrary, the function **mapcan** in Lisp is a destructive operation even in its specification. It applies a function to all the elements of the list which is an argument of **mapcan**, and concatenates the resulting lists destructively. For example, when evaluating **(double '(a b c))** shown in Section 3.3, the function **#'(lambda (h) `(,h ,h)) x)** is applied to the elements **a**, **b** and **c** and it returns **(a a)**, **(b b)** and **(c c)** respectively. These resulting lists are destructively concatenated and **mapcan** and **double** returns the list **'(a a b b c c)**.

Though predicate **mapcan** and function **mapcan** are different in specification, the former can be used for most of the purposes that the latter is used such as the above example. This functionality of predicate **mapcan** is also enabled by the logical variable.

4.2 Termination conditions in the do construct

The termination condition is described separately from the "loop body" in a **do** construct. This feature is explained along with its good points below.

For example, in **fac1** in Section 2, the final value of the "control variable", **X**, is given in an *actual* parameter of **do**. The termination condition is that the value of **X** calculated in the loop is unifiable with this final value. On the contrary, termination conditions are usually written in the *definition* of the recursive predicate when using recursion. For example, in **fac2**, the final value, **0**, is given as a *formal* parameter in the first clause of **fac2**.

In comparison to a normal termination condition of recursion, writing a termination condition outside of the loop body has two major good points.

The first good point is readability. A reader of the program can find the termination condition more easily if it is written in a **do** construct. Though one can find the termination condition easily in a well-formulated recursive program, programs are not easily well-written using recursion. That means that loops are written in more uniform and readable way using **do**, than using recursions.

The second good point is that the loop body is more generally usable than a normal recursive program because it can be used with different termination conditions from the original one. For example, if **X** is a list and **X1** is a sublist of **X**, partial list operations can be easily written as follows: if **X-[]** in the right-hand side of **reverse** is replaced by **X-X1**, the partial list is reversed. If **X-[]** in the right-hand side of **length** is replaced by d-list **X-X1**, the length of this d-list is computed. On the contrary, termination conditions are usually hard-wired in recursive predicate definitions, so it is difficult to use them in other purposes.

4.3 Termination conditions outside of the do construct

The termination condition can even be written outside of a **do** construct. This feature is explained along with the merits and demerits below.

If termination conditions could not be written outside of the **do** construct, unification would be the only possible operation with which to test them. However, any predicate can be used for testing them outside of **do**. Even termination conditions can be supplied by user. For example, with respect to Newton's method in the example in Section 2, predicate '**<**' is used for testing, and the precision of the solution can be easily changed by changing the termination condition **Eps<1e-5**. The termination can also be controlled by user. In fact, the program in Section 2 can be used as follows. A user can type **?-newton(X)**. and will get a result. If it is not satisfiable in precision, he or she can type '**;**' (requires backtracking) and will get a more precise result.

This feature has both good and bad points. It is obvious from the above explanation that termination conditions have powerful describability. However, they are also unstructured and unreadable. Extensive use of termination conditions outside of loops results in unstructured loops, thus loosing the main benefit of the **do** construct, namely readability of programs.

A functional refinement of predicate **do** will clear up this problem. The following is an example of a refined version of **do** and its usage in Newton's method. An additional predicate for the termination test is given to **do**.

```
do(P,C,A1-A1,A2-A2) :- apply(C,[A1,A2]).
do(P,C,A1-A1e,A2-A2e) :-
   apply(P,[A1-A1a,A2-A2a]), do(P,A1a-A1e,A2a-A2e).

newton(X) :- do(newton1, [_,Eps]\Eps<1e-5, 1-X, 999-_).
```

Predicate **newton1** is the same as in Section 2. The λ term given as the second parameter of **do** tests the termination condition.

4.4 Efficiency

The implementation of **do** shown in Section 3 is inefficient because the "loop body" is executed interpretively. However, it is easy to compile **do** constructs into normal loops. There are two methods to compile as such. The first method is to compile **do** constructs into loops in machine code or a procedural language directly by a specialized Prolog compiler. The second method is to expand **do** and **apply** and translate them into tail recursions in Prolog, and then to compile them into loops using a Prolog compiler that performs tail-recursion optimization. The second method is much easier because there is no need to develop a new Prolog compiler because predicates using **do** and **apply** can be expanded into normal Prolog predicates and no syntactic and semantic extension are necessary. The object code is as efficient as, or more efficient than code generated from recursions in either method.

5. Conclusion

This paper presented a general purpose iterative predicate **do** and the extended λ term. Their combination enables a programmer to write most iterative control structures, such as arithmetical iterations, **append**, **member** or **mapcar**, in more readable way and to make them more general-purpose without loosing their efficiency. Unification and logical variables

in Prolog enables some extensive uses of the control structures compared to those of other programming languages such as Lisp. It would be better to extend Prolog to include **do** or its refined version into its language specification for the sake of better program development.

References

[Dod 83] Dodson, D. C., and Rector, A. L.: "LOGAL" : Algorithmic Control Structures for Prolog, *Eighth International Conference on Artificial Intelligence*, pp. 536-538, 1983.

[Kon 87] Kondoh, S., and Chikayama, T.: Macro Processing in Prolog, *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 466-480, 1987.

[Mun 86] Mukai, T.: Procedurally Oriented Programming Techniques in Prolog, *IEEE Expert*, Vol. 1, No. 2, pp. 41-47, 1986.

[Nad 88] Nadathur, G., and Miller, D.: An Overview of λ Prolog, *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 810-827, 1988.

Appendix: An implementation of `apply`

An interpretive implementation of **`apply`** is shown below:

```
apply(Lambda\Body,Args) :- !,
   s_subst(Args,Lambda,Body,Goal), Goal.
apply((Term1,Term2),Args) :- !,
   apply(Term1,Args), apply(Term2,Args).
apply((Term1;Term2),Args) :- !,
   (apply(Term1,Args); apply(Term2,Args)).
apply(\+Term,Args) :- !, \+apply(Term,Args).
apply(Term,Args2) :-
   Term=..[F|Args1], append(Args1,Args2,Args),
   Goal=..[F|Args], Goal.
```

The first clause is for the λ term application. Logical variables in the λ term are renamed by predicate **`s_subst`**. The implementation of predicate **s_subst** is shown below. The second clause of **`apply`** is for *and-composition*, and the third for *or-composition*. The fourth clause is for negation. The last clause is for named predicate application, but its function is extended. The first parameter, **`Term`**, can be an atom or a functor. If it is an atom, it is applied in the same manner as described in Section 2. If it is a functor, its arguments are regarded as "constant parameters" for the predicate. For example, **`mapcar(append([a]),[b,c],Y)`** results in **`Y=[[a,b],[a,c]]`**. That means list **`[a]`** is supplied as the first parameter for all the application of **`append`**.

A simple but inefficient and ugly implementation of **`s_subst`** is as follows:

```
s_subst(N,O,OStruct,NStruct) :-
   asserta(s_subst_(O,OStruct)),
   retract(s_subst_(N,NStruct)), !.
```

The above predicate **s_subst** uses **assert** and **retract**. All the variables in the λ term are renamed in this simple implementation. It is sometimes inconvenient and it will be better not to rename variables that do not appear in lambda list. The above implementation is also probably slower because of **assert** and **retract**. Thus, more sophisticated implementation without using **assert** and **retract** will be preferred.