A Simple Prolog Techniques Editor for Novice Users[1]

Dave Robertson
Department of Artificial Intelligence,
University of Edinburgh.
22nd March 1991

Abstract

This paper describes a working prototype system which uses descriptions of standard Prolog techniques to provide a basic techniques editing system, ultimately intended for use by novice programmers. A notation for representing techniques, based on Definite Clause Grammars, is described in the context of previous theoretical work by Kirschenbaum, Lakhotia and Sterling. Details are supplied of a mechanism for using these techniques to provide guidance during program construction and an example is provided of the system in operation. I conclude by suggesting the extensions needed in order to make the prototype useful for practical applications.

1 Introduction

Many people have expressed an interest in supplying software tools which will make Prolog programming easier and/or more efficient. Most of the effort of implementation seems to have been directed at the debugging phase and there seem to be very few implementations of systems which provide help with the initial construction of programs. This seems surprising to me because when I teach Prolog I spend a lot of time making sure that students have grasped the basics of writing Prolog code, while the amount of time I spend on debugging is comparatively modest. The way I normally teach basic Prolog programming is as follows : I describe certain simple techniques which are generally useful (*e.g.* basic deconstruction of a list) ; I show how these can be applied for simple examples ; and finally I get the students to apply these same techniques to other examples. This arrangement seems to work but it takes up a great deal of my time because I have to keep going over examples until each student has understood how the techniques are used. If a computer program could be written which would be able to guide students in the application of these techniques to sets of example problems then I would be spared the monotony of working through the same tired set of examples each year and students would have easier access to help in their initial attempts to write Prolog code. This paper describes a first attempt at constructing this sort of program.

[1]A shortened version of this paper appears in the proceedings of the Association for Logic Programming–UK 3rd Annual Conference, held at the University of Edinburgh from 10–12 April 1991.

2 Representing Basic Methods of Constructing Prolog Programs

A recent paper by Kirschenbaum, Lakhotia and Sterling ([Kirschenbaum *et al.* 89]) seems to present a useful framework for a Prolog techniques tutor. A related technique has been described in [Plummer 90] which gives details of an implementation but [Kirschenbaum *et al.* 89] provides a more convincing analysis of the problem. Gegg–Harrison, has produced generalisation hierarchies for some Prolog list processing techniques ([Gegg-Harrison 89]) and has suggested that these could be used in a similar way to the LISP cliche libraries of the Programmers Apprentice. Although these papers are interesting from a theoretical point of view, they do not address in detail the practical problems of *using* the idea of techniques to support program construction.

My experiments have been based on the view of program construction promoted in [Kirschenbaum *et al.* 89]. This may be summarised as follows:

- When constructing a program we start out with a basic plan of the control flow of the program. This basic plan is referred to as a *skeleton*.
- We also have available various standard methods for performing useful tasks, such as passing back results from computations. These are referred to as *techniques*.
- A technique may be applied to a skeleton to obtain an *extension* of the skeleton.
- Extensions may be composed to produce completed programs.

In this section I shall provide a notation in which skeletons and techniques may be represented in Prolog. Then, in Section 3, I shall describe how these may be used as the basis for a simple techniques editor.

2.1 Skeletons

Skeletons provide the basic flow of control which will be the starting point for program development. One of the simpler skeletons would be list traversal, where a list is deconstructed by removing head elements until the empty list is reached. In Prolog–like notation this might be written as follows:

$$
\begin{array}{l}
PredName([]). \\
PredName([H|T]) :- \\
\quad\quad Test_1, \\
\quad\quad PredName(T). \\
PredName([H|T]) :- \\
\quad\quad Test_2, \\
\quad\quad PredName(T). \\
\quad\quad \vdots \\
PredName([H|T]) :- \\
\quad\quad Test_N, \\
\quad\quad PredName(T).
\end{array}
$$

where $Test_1 \cdots Test_N$ are tests which determine which of the clauses to use for deconstructing the list. The degenerate case of this skeleton occurs when there is no need to use any tests to distinguish the different clauses used to deconstruct the list, in which case we have simply:

$$
\begin{array}{l}
PredName([]). \\
PredName([H|T]) :- PredName(T).
\end{array}
$$

The problem is to provide a notation which will allow us the flexibility of applying the same basic skeleton, parameterised by the particular cases which it needs to cover, to obtain an instantiation for some particular problem. DCG's offer a useful mechanism for performing this task.

Agree to represent each goal of a clause as a list with the first element being the predicate name and subsequent elements being the arguments, in order. Also agree to represent each clause as a list of goals with the first element being the head of the clause and subsequent elements being the body of the clause. A DCG rule for a skeleton is defined using the notation:

$$skeleton(Type, PredName) \rightarrow Defn$$

where $Type$ defines the type of skeleton; $PredName$ gives the predicate name for the procedure to which the skeleton applies; and $Defn$ will contain a definition of a valid sequence of clauses for the skeleton. The $Type$ argument is used not only to distinguish between different varieties of skeleton but also to provide parameters used in controlling the instantiation of the skeleton. For example, the list traversal skeleton mentioned earlier must be parameterised with the test cases which need to be used in order to distinguish between the clauses for deconstructing the list so the $Type$ argument in this case is: $traverse([Test_1, \cdots Test_N])$. The DCG definition for this skeleton then becomes:

$skeleton(traverse([]), PredName) \rightarrow$
$\quad [\ [[PredName, [\_|T]], [PredName, T]],$
$\quad\ \ [[PredName, []]]].$
$skeleton(traverse([Case|RestCases]), PredName) \rightarrow$
$\quad traverse\_cases(PredName, [Case|RestCases]).$

$traverse\_cases(PredName, []) \rightarrow [[[PredName, []]]].$
$traverse\_cases(PredName, [case([H|T], Test)|RestCases]) \rightarrow$
$\quad [[[PredName, [H|T]], Test, [PredName, T]]],$
$\quad traverse\_cases(PredName, RestCases).$

where the first clause for $skeleton/4$ takes care of the degenerate case where there are no test cases for differentiating between recursive clauses. The second clause for $skeleton/4$ deals with the case where there are some test cases and it generates the appropriate recursive clauses using $traverse\_cases/4$.

An example of the use of this skeleton would be where we wanted to construct a program named get_evens which collected all the even numbers in some list of numbers. There are two test cases which distinguish between the recursive cases of this program: depending on whether the head of the list is odd or even. Thus the DCG for the $traverse$ skeleton is called as follows:

$|\ \ ?-\ \ skeleton(\ traverse([case([H1|T1], even(H1)), case([H2|T2], \backslash + even(H2))]),$
$\qquad\qquad\qquad get\_evens, S, []).$

and would return the instantiation of the skeleton as:

$[\ [[get\_evens, [H1|T1]], even(H1), [get\_evens, T1]],$
$\ \ [[get\_evens, [H2|T2]], \backslash + even(H2), [get\_evens, T2]],$
$\ \ [[get\_evens, []]]]$

which corresponds to the Prolog program:

$get\_evens([H1|T1]) :- even(H1), get\_evens(T1).$
$get\_evens([H2|T2]) :- \backslash + even(H2), get\_evens(T2).$
$get\_evens([]).$

Of course, this isn't the completed program because, although the control of flow is correct for the get_evens procedure, there remains the problem of accumulating and passing back the completed list of even numbers. To solve this problem we must be able to represent techniques.

2.2 Techniques

A skeleton provides an initial sequence of clauses which forms the basis for the rest of the program. Techniques must augment these sequences with extra arguments and subgoals necessary to complete the program. These techniques should be general, in the sense that they will apply to any sequence of clauses for which they are valid. It is tempting to construct techniques simply as standard Prolog procedures which unpack clause sequences and add new elements as necessary. However, this is unsatisfactory because the techniques are quite complex to implement directly in Prolog and this makes it tricky to construct and debug the techniques. What is needed is a more specialised (though not constraining) notation which is specially designed to support the translation process, in the same way that the DCG notation is particularly suited to grammar processing.

One way of tackling the translation process is to "unpack" the initial sequence of clauses using a standard DCG approach but at each step in the unpacking to map the element of the initial sequence which is currently being considered onto a translated version in the augmented sequence. The idea is that the parsing of the initial sequence controls the generation of the augmented version. Since this involves processing two DCGs "simultaneously" we need some new notation which will represent the appropriate operations:

- A mapping is denoted by the term: $MapName :: Body$, where $MapName$ is a Prolog term with 0 or more arguments and $Body$ is a sequence of operations defined informally as follows:
  - The operation $Seq1 \Rightarrow Seq2$, which maps a subsequence of the initial list to a subsequence of the new list.
  - The operation $Seq1 \Rrightarrow Seq2$, which maps a subsequence of the initial list to an element of the new list.
  - A reference to some subsidiary mapping definition.
- A subsequence of a list can be represented as either:
  - A list of terminal symbols: $[S_1, \cdots, S_N]$.
  - An empty sequence symbol: [].
  - A combination of lists of terminal symbols and sequences of procedures: $\{P_1, \cdots, P_M\}$.
  - A reference to some subsidiary mapping definition, provided that the subsequence appears on the right hand side of a mapping operator.

As an introduction to the way in which this notation is used, consider the problem of flattening a list. There are many ways of doing this in Prolog – the simplest involving the recursive application of the flattening procedure to sublists and subsequent appending of flattened sublists to the rest of the processed list. In terms of techniques this may be thought of as a parse of the input (nested) list and a mapping of each element of the input list onto a subsequence of the output (flattened) list. The definition for this procedure is:

$$
\begin{aligned}
&flatten :: [[\_|\_]] \Rightarrow flatten,\\
&\qquad flatten.\\
&flatten :: ([A], \{atom(A)\}) \Rightarrow [A],\\
&\qquad flatten.\\
&flatten :: [\,] \Rightarrow [\,].
\end{aligned}
$$

The first clause of this definition states that if the first element of the current input sequence is a list then that list is mapped to the output sequence by applying the $flatten$ mapping to it. The second clause states that if the first element is an atom then it is mapped directly across as the first element of the output sequence. The third clause states that if the input sequence contains no elements then the output sequence is also empty.

To turn this notation into computable procedures for constructing mappings between lists it is necessary to provide a compilation mechanism which adds in the necessary difference lists (in a manner analogous to the compilation of DCG rules in standard Prolog). There is little point in boring the reader with the details of this mechanism. It is sufficient to show the Prolog code which is produced by compiling the $flatten$ definition above.

$$
\begin{aligned}
&flatten(A, B, C, D) :-\\
&\qquad 'C'(A, [E|F], G), flatten([E|F], [\,], C, H),\\
&\qquad flatten(G, B, H, D).\\
&flatten(A, B, C, D) :-\\
&\qquad ('C'(A, E, F), atom(E)), 'C'(C, E, G),\\
&\qquad flatten(F, B, G, D).\\
&flatten(A, A, B, B).
\end{aligned}
$$

This compiled Prolog version of $flatten$ has 4 arguments: the input sequence, A ; the remainder of the input sequence after parsing, B ; the mapped sequence, C ; and the remainder of the elements in the mapped sequence, D. The $'C'/3$ predicate is the standard Prolog predicate for removing terminal elements from DCGs and is defined as:

$$'C'([X|T], X, T).$$

Since we typically require that there should be no remaining (unparsed) elements in either the input or mapped sequences, these arguments (B and D in the example) are normally instantiated to the empty list when the procedure is called. Thus, to flatten the list $[[a,b],[c,[d]]]$ using the above procedure we would give the goal:

$$| \ ?- \ flatten([[a,b],[c,[d]]],[],X,[]).$$

which would bind X to $[a,b,c,d]$.

Let us now consider how this mapping mechanism may be used to implement techniques for Prolog programs. Again, this explanation is easier to understand in the context of a specific example. Consider the problem of adding a simple accumulator to a recursive program. For each recursive clause, we need to add an extra argument to the head and recursive subgoal to pass back the result and, in addition, we must ensure that the final value obtained when the recursive clause succeeds is updated with whatever structure we intend to accumulate. Expressing this using the notation for techniques we obtain:

$technique(back\_accumulate, \_) ::$
$\quad [Head] \Rrightarrow add\_args([Result]),$
$\quad non\_recursive\_seq(Head),$
$\quad ([R], \{recursive\_subgoal(Head, R)\}) \Rrightarrow add\_args([PartialResult]),$
$\quad [] \Rightarrow ['\$'(update\_value(PartialResult, Result))].$
$technique(back\_accumulate, \_) ::$
$\quad [Head] \Rrightarrow add\_args([Result]),$
$\quad non\_recursive\_seq(Head),$
$\quad [] \Rightarrow ['\$'(instantiate(Result))].$

This technique is named *back_accumulate*. Its first mapping definition deals with the recursive case: adding an extra *Result* argument to the head of the clause; passing through some non–recursive sequence; then finding the recursive subgoal and adding an extra *PartialResult* argument to it; finally adding an extra subgoal for obtaining the value of *Result* from *PartialResult*. The second mapping definition deals with the non–recursive case: adding an extra *Result* argument to the head of the clause and inserting an extra subgoal to instantiate *Result* for a "base" value. Note that, since we require a general definition of this form of accumulation, it is not possible to say precisely how the final value of *Result* will be obtained. Thus the predicates $update\_value(PartialResult, Result)$ and $instantiate(Result)$ do not refer to particular Prolog procedures but, instead, provide a "flag" that procedures of this form are required in the code at those points. To distinguish these flags from object level code they are enclosed within a $\$/1$ predicate. The instantiation of these procedures is dealt with in Section 2.3.

Appendix A provides definitions of the mappings for $non\_recursive\_seq/5$ and $add\_args/5$.

To show how this definition is used, recall the partial definition of get_evens which was produced by applying the $traverse$ skeleton in Section 2.1. An accumulator is required in this definition in order to record the even numbers which are found when deconstructing the list of numbers. we can perform this addition by applying the $back\_accumulate$ mapping definition to each clause of get_evens, in which case we obtain the list of sequences:

$$
\begin{array}{l}
[\ [[get\_evens, [H1|T1], Result1], \\
\qquad\qquad even(H1), \\
\qquad\qquad [get\_evens, T1, PartialResult1], \\
\qquad\qquad '\$'(update\_value(PartialResult1, Result1))], \\
\ \ [[get\_evens, [H2|T2], Result2], \\
\qquad\qquad \backslash +\, even(H2), \\
\qquad\qquad [get\_evens, T2, PartialResult2], \\
\qquad\qquad '\$'(update\_value(PartialResult2, Result2))], \\
\ \ [[get\_evens, [], Result3], \\
\qquad\qquad '\$'(instantiate(Result3))]]
\end{array}
$$

which corresponds to the Prolog program:

$$
\begin{array}{l}
get\_evens([H1|T1], Result1) :- \\
\qquad even(H1), \\
\qquad get\_evens(T1, PartialResult1), \\
\qquad '\$'(update\_value(PartialResult1, Result1)). \\
get\_evens([H2|T2], Result2) :- \\
\qquad \backslash +\, even(H2), \\
\qquad get\_evens(T2, PartialResult2), \\
\qquad '\$'(update\_value(PartialResult2, Result2)). \\
get\_evens([], Result3) :- \\
\qquad '\$'(instantiate(Result3)).
\end{array}
$$

2.3 Elaborations

The final component of the techniques editing system is a means of elaborating upon the general procedures flagged with a \$ symbol in the partially developed programs. This is a different sort of activity from that of applying techniques because techniques are mappings which are applied across all clauses of a program, while an elaboration will apply to a single clause. This being so, it is still possible to implement elaborations using the same mapping procedure that was used in Section 2.2. A general purpose mapping definition for instantiating $\$/1$ procedures is shown below:

$elaboration(inst\_subproc(Proc, Inst), \_) ::$
$\quad 'SEQ',$
$\quad ['\$'(Proc)] \Rightarrow [Inst],$
$\quad 'SEQ'.$

where $'SEQ'$ simply maps a segment of the input list to the output list (see Appendix A). Of course, all this does is to replace the "flagged" procedure ($Proc$) with the pre–supplied instantiation of the procedure ($Inst$). This leaves the difficult job of deciding what the code supplied for $Inst$ should be. We return to this topic in Section 3. In the meantime, it is possible to apply the elaboration mapping to the running example of the get_evens program in order to obtain a runnable program. Three elaborations are necessary:

Flagged subgoal	Elaboration
$'\$'(update\_value(PartialResult1, Result1))$	$Result1 = [H1 \mid PartialResult1]$
$'\$'(update\_value(PartialResult2, Result2))$	$Result2 = PartialResult2$
$'\$'(instantiate(Result3))$	$Result3 = []$

This produces the program:

$get\_evens([H1|T1], Result1) :-$
$\quad even(H1),$
$\quad get\_evens(T1, PartialResult1),$
$\quad Result1 = [H1|PartialResult1].$
$get\_evens([H2|T2], Result2) :-$
$\quad \backslash + \ even(H2),$
$\quad get\_evens(T2, PartialResult2),$
$\quad Result2 = PartialResult2.$
$get\_evens([], Result3) :-$
$\quad Result3 = [].$

Which can be simplified by removing the explicit unification subgoals to form:

$get\_evens([H1|T1], [H1|PartialResult1]) :-$
$\quad even(H1),$
$\quad get\_evens(T1, PartialResult1).$
$get\_evens([H2|T2], PartialResult2) :-$
$\quad \backslash + \ even(H2),$
$\quad get\_evens(T2, PartialResult2).$
$get\_evens([], []).$

In this section I have described a way of constructing simple Prolog programs, based on notions of skeletons and techniques described in [Kirschenbaum *et al.* 89]. It is worth noting in passing that their method of application of techniques is different from the method which I have used. They construct programs by applying techniques independently to a skeleton, thus obtaining several program components, and finally merge these components together to obtain a completed program. I retain a single partial program and successively apply each technique/elaboration to it to obtain a linear sequence of development. Both approaches rely on the limiting assumption that the application of techniques doesn't affect the flow of control of the program (as dictated by the skeleton). In the next section I will show how these mechanisms can be employed to produce a very limited form of automation of the template application process.

3 A Simple Techniques Editor

The program described in this section isn't intended to be a final, polished piece of software. It is merely a simple prototype, designed to show roughly how an interface might be constructed around the basic template application mechanism. The workings of the system will be described with reference to a sequence of screen "snapshots" obtained during the development of the standard *append*/3 predicate.

The system is provided with a library of descriptions of predicates which the user might want to define. Each description provides: the name of the predicate ; an ordered list of its arguments with the type of each argument recorded ; and a list of test goals for which the predicate must succeed. For the *append* example this information would be:

- Its name is *append*.
- Its arguments are $[a:1 = list, a:2 = list, a:3 = list]$. Where $a:1$, $a:2$ and $a:3$ identify the arguments of the predicate and each argument is of type *list*.
- The single test goal is $(append([a, b], [c, d], R),\ R == [a, b, c, d])$.

The list of test goals is displayed to the user as a menu from which the user selects the *append* predicate, as shown below.

```
1 : append [a:1=list,a:2=list,a:3=list]
2 : length [a:1=list,a:2=number]
3 : evens [a:1=list,a:2=list]
4 : member [a:1=list,a:2=argument]
Choose an item from the menu: 1.
```

The next move is to obtain a choice of skeleton to be used for controlling the flow of execution of the program. The system currently contains three skeletons: for the complete traversal of a list; for a short traversal of part of a list ; and for a search which may or may not traverse the entire list. To provide the system with a means of narrowing down the selection of appropriate skeletons (assuming that there will be a wide choice at some point in the future) each skeleton is provided with a set of preconditions which must be established in order for it to be valid for a particular problem. For example, the preconditions for a list traversal skeleton are:

- The predicate must have an argument which is of type list.
- The problem must involve the processing of all elements in the list.
- The test cases necessary to distinguish between recursive cases of the skeleton must be known (see Section 2.1).

The first of the above preconditions may be satisfied easily from the initial predicate specification but the other two preconditions require more sophisticated information. Currently, this information is also stored along with the initial predicate specification. Thus for the *append* predicate the system is prepared with the information that all elements of the list must be processed and that there is no need to distinguish between different recursive cases. If the system were to be extended it would be necessary to provide a more sophisticated method for obtaining ancillary components of the problem description from the user, rather than "precanning" them for each example. This is a similar sort of problem to that addressed in the EL system ([Robertson *et al.* 91]).

The result of this process, in the case of *append*/3, is to isolate a single skeleton, which the user chooses.

```
Pick a skeleton from the menu below:
1 : traverse([]),[a:2=list,a:3=list]
Select a skeleton :  1.
```

The system now begins the process of technique application. The display (below) shows the predicate name and its arguments, with an `X` marking those arguments which already have been accounted for. Below this is a display of the partial program obtained by application of the *traverse* skeleton. The system then generates a menu of potential techniques which might be applied to the current partial program. The user chooses the *accumulate* technique, which is the standard 2–variable type of accumulation (see Appendix A).

```
-------- You have accounted for these arguments: -------
append(X=list, a:2=list, a:3=list).
---------------- The program so far is: ----------------
Clause 1 : append([A|B]):-
                    append(B)

Clause 2 : append([])

---------------------------------------------------
1 : back_accumulate
2 : add_carrier
3 : sum
4 : accumulate
Select a technique :  4.
```

The program resulting from the application of this template is shown in the next screen display. Note that this contains a "flagged" requirement for a procedure to obtain the value of the variable E from A. The system generates a suggestion of how this flagged procedure might be elaborated and the user (erroneously) accepts this suggestion.

```
Clause 1 : append([A|B],C,D):-
                    PROCEDURE : Obtain the value of E from A,
                    append(B,[E|C],D)

Clause 2 : append([],F,F)

---------------------------------------------------
TO CLAUSE 1 :

1 : DEFINE : Obtain the value of E from A AS : E=A
Select an elaboration :  1.
```

There is now no further scope for extending the program – all the arguments have been accounted for and there are no unspecified subgoals. The system tests whether the completed program can satisfy the test goals given in its original specification. It cannot so the system displays the current program and gives the user the opportunity to backtrack through previous stages of template/skeleton application.

```
---------------- The program so far is: ----------------
Clause 1 : append([A|B],C,D):-
                    E=A,
 append(B,[E|C],D)

Clause 2 : append([],F,F)

--------------------------------------------------
Unfortunately, your program doesn't work.
You will have to backtrack to a previous stage of development
and restart from there.

Type any key to continue:
```

One step back isn't enough because the problem is in the template, not the elaboration.

```
Clause 1 : append([A|B],C,D):-
                    PROCEDURE : Obtain the value of E from A,
                    append(B,[E|C],D)

Clause 2 : append([],F,F)

--------------------------------------------------
TO CLAUSE 1 :

1 : DEFINE : Obtain the value of E from A AS : E=A
Select an elaboration :  back.
```

Now there is a chance to apply a new template...but which ? The user has the option of showing the effect of applying a particular template without applying the template permanently. He decides to look at the *back_accumulate* template and the program which would result from the application of this template is shown immediately below the prompt. This is not what is required because the second argument should carry the second input list, not accumulate some new list. Therefore the user decides to apply the *add_carrier* technique.

```
-------- You have accounted for these arguments: -------
append(X=list, a:2=list, a:3=list).
---------------- The program so far is: ----------------
Clause 1 : append([A|B]):-
                    append(B)

Clause 2 : append([])

--------------------------------------------------
1 : back_accumulate
2 : add_carrier
3 : sum
4 : accumulate
Select a technique :  show(1).
Clause 1 : append([A|B],C):-
                    append(B,D),
                    PROCEDURE : Obtain the value of C from D

Clause 2 : append([],E):-
                    PROCEDURE : Instantiate E

Select a technique :  2.
```

The *back_accumulate* technique is then applied to the third argument.

```
-------- You have accounted for these arguments: -------
append(X=list, X=list, a:3=list).
---------------- The program so far is: ----------------
Clause 1 : append([A|B],C):-
                    append(B,C)

Clause 2 : append([],D)

--------------------------------------------------
1 : back_accumulate
2 : add_carrier
3 : sum
4 : accumulate
Select a technique :  1.
```

Now it is necessary to elaborate on the flagged subgoal in the first clause. The user decides to do this by inserting $D = [A|E]$ at that point in the program.

```
Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    PROCEDURE : Obtain the value of D from E

Clause 2 : append([],F,G):-
                    PROCEDURE : Instantiate G

--------------------------------------------------
TO CLAUSE 1 :

1 : DEFINE : Obtain the value of D from E AS : D=[C|E]
2 : DEFINE : Obtain the value of D from E AS : D=[B|E]
3 : DEFINE : Obtain the value of D from E AS : D=[C|E]
4 : DEFINE : Obtain the value of D from E AS : D=[B|E]
5 : DEFINE : Obtain the value of D from E AS : D=[A|E]
6 : DEFINE : Obtain the value of D from E AS : D=E
Select an elaboration :  5.
```

The system now prompts for an elaboration of the only remaining flagged subgoal. However, the user decides first to take a look at the history of development of the program.

```
Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    D=[A|E]

Clause 2 : append([],F,G):-
                    PROCEDURE : Instantiate G

--------------------------------------------------
TO CLAUSE 2 :

1 : DEFINE : Instantiate G AS : G=[]
2 : DEFINE : Instantiate G AS : G=F
Select an elaboration :  replay.
```

The system replays the sequence of steps in chronological order, starting with the application of the initial skeleton.

```
The first step was to apply the skeleton:
traverse([])
to obtain the code shown below:
--------------------------------------------------
Clause 1 : append([A|B]):-
                    append(B)

Clause 2 : append([])

The technique:
add_carrier
was applied to obtain the code shown below:
--------------------------------------------------
Clause 1 : append([A|B],C):-
                    append(B,C)

Clause 2 : append([],D)

The technique:
back_accumulate
was applied to obtain the code shown below:
--------------------------------------------------
Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    PROCEDURE : Obtain the value of D from E

Clause 2 : append([],F,G):-
                    PROCEDURE : Instantiate G

The elaboration:
DEFINE : Obtain the value of D from E AS : D=[A|E]
was applied to obtain the code shown below:
--------------------------------------------------
Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    D=[A|E]

Clause 2 : append([],F,G):-
                    PROCEDURE : Instantiate G

Type any key to continue:
```

The user then instantiates the final flagged subgoal to $G = F$.

```
Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    D=[A|E]

Clause 2 : append([],F,G):-
                    PROCEDURE : Instantiate G

--------------------------------------------------
TO CLAUSE 2 :

1 : DEFINE : Instantiate G AS : G=[]
2 : DEFINE : Instantiate G AS : G=F
Select an elaboration :  2.
```

The program is now complete and runnable so the system displays the final result and simplifies it to get rid of the explicit unification subgoals.

```
Congratulations. You have written a program which works.
This is what you ended up with:

Clause 1 : append([A|B],C,D):-
                    append(B,C,E),
                    D=[A|E]

Clause 2 : append([],F,G):-
                    G=F


This program can be simplified to give the program:

Clause 1 : append([A|B],C,[A|D]):-
                    append(B,C,D)

Clause 2 : append([],E,E)
```

4 Conclusions and Future Work

Section 2 describes a method for representing Prolog techniques using a notation based on DCGs. In Section 3 this notation is used to provide a system which allows the application of techniques by users with little knowledge of Prolog. However, this prototype is far from being able to provide the level of guidance necessary to support novice users. Many improvements to the interface could be made but these are subsidiary to more basic work which is required to extend the functionality of the system. Some of the areas which require improvement are:

The range of skeletons, techniques and extensions: Only a tiny number of these are available in the current system. More work is required to collect a larger number of templates and to organise them into a coherent package. A significant amount of work has already been done by other researchers in classifying techniques (*e.g.* [Gegg-Harrison 89], [Kirschenbaum *et al.* 89],[Brna *et al.* 91]) but this needs to be standardised and implemented within a uniform computational framework.

The range of example programs: At present, the system restricts the user's choice of programs to a small number of standard examples which it can construct automatically. The advantage of imposing this restriction is that the system can construct complete programs automatically for these examples and so can complete a program if the user gets stuck. If the user were allowed to provide his/her own examples then this guarantee would no longer apply. Work is required to extend the range of initial examples (a comparatively easy task) and to provide some way of allowing users to specify realistic problems of their own (in general, an extremely difficult task).

Permitting more complex programs: As it stands, the system can only cope with definitions of single predicates . It cannot construct a program which requires two or more interacting predicates. This is a major limitation which needs to be lifted. The main problem would be the increase in search space which would ensue if sub–procedures were permitted. This creates a greater requirement for tight controls on the range of techniques which could be applied (see next item).

Describing the problem which the program must solve: Ancillary information is required whenever skeletons, techniques or elaborations are applied. This information is sometimes readily available by examination of the structure of the partial program but at other times the information has to do with the general nature of the problem which has to be solved. To select the *traverse* skeleton for the *append* example it was necessary for the system to be told that all elements of the list needed to be processed and that there was no need to distinguish between different recursive cases. Currently, this information is "precanned" in the system and is hidden from the user. This is highly undesirable because users must be aware of the conditions under which various techniques are applied. The architecture which suggests itself is that used in the ECO project ([Robertson *et al.* 91]), where a "high level" problem description was used to capture knowledge about the problem which had to be solved and this was used to control generation of the program.

References

[Brna *et al.* 91] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, D. Robertson, B. Smith, and M. van Someren. Prolog programming techniques. *Instructional Science*, 20(2/3), 1991.

[Gegg-Harrison 89] T.S. Gegg-Harrison. Basic prolog schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University, September 1989.

[Kirschenbaum *et al.* 89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for prolog programming. Tr 89-170, Case Western Reserve University, 1989.

[Plummer 90] D. Plummer. Cliche programming in prolog. In *Proceedings of the META-90 workshop*, Leuven, Belgium, 1990. META-90.

[Robertson *et al.* 91] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M Uschold. *Eco-Logic: Logic-Based Approaches to Ecological Modelling.* MIT Press (Logic Programming Series), 1991. ISBN 0-262-18143-6.

A Utility Mapping Definitions

A direct mapping between input and output sequences:

$$
\begin{array}{l}
{'SEQ'} :: \\
\qquad [] \Rightarrow []. \\
{'SEQ'} :: \\
\qquad [X] \Rightarrow [X], \\
\qquad {'SEQ'}.
\end{array}
$$

Accumulation on the way down through a recursion:

$$
\begin{array}{l}
technique(accumulate, Clause) :: \\
\qquad [Head] \Rrightarrow add\_args([Sofar, Final]), \\
\qquad \{deconstructed\_arg(Head, Clause, H)\}, \\
\qquad [] \Rightarrow ['\$'(update\_value(H, Value))], \\
\qquad recursion\_accumulate(Head, [Value|Sofar], Final). \\
technique(accumulate, \_) :: \\
\qquad [\_] \Rrightarrow add\_args([Final, Final]).
\end{array}
$$

$$recursion\_accumulate(Head, Sofar, Final) ::$$

$([R], \{recursive\_subgoal(Head, R)\}) \Rrightarrow add\_args([Sofar, Final]),$
$non\_recursive\_seq(Head),$
$\{!\}.$
$recursion\_accumulate(Head, Sofar, Final) ::$
$([R], \{recursive\_subgoal(Head, R)\}) \Rrightarrow add\_args([Sofar, NewSofar]),$
$recursion\_accumulate(Head, NewSofar, Final).$
$recursion\_accumulate(Head, Sofar, Final) ::$
$([R], \{\backslash + recursive\_subgoal(Head, R)\}) \Rightarrow [R],$
$recursion\_accumulate(Head, Sofar, Final).$

$non\_recursive\_seq(Goal) ::$
$([R], \{\backslash + recursive\_subgoal(Goal, R)\}) \Rightarrow [R],$
$non\_recursive\_seq(Goal).$
$non\_recursive\_seq(\_) ::$
$[] \Rightarrow [].$

Maps input list to a list with $Args$ tacked to the end:

$add\_args(Args) ::$
$[Functor] \Rightarrow [Functor],$
$'SEQ',$
$[] \Rightarrow Args.$

A direct mapping between non–recursive input and output sequences:

$non\_recursive\_seq(Goal) ::$
$([R], \{\backslash + recursive\_subgoal(Goal, R)\}) \Rightarrow [R],$
$non\_recursive\_seq(Goal).$
$non\_recursive\_seq(\_) ::$
$[] \Rightarrow [].$