Applying Prolog Programming Techniques

A. Bowles, D. Robertson, W. Vasconcelos, M.Vargas-Vera, D. Bental,
Department of Artificial Intelligence,
University of Edinburgh,
80 South Bridge,
Edinburgh, EH1 1HN.
email: andy@aisb.ed.ac.uk

Applying Prolog Programming Techniques

A. Bowles, D. Robertson, W. Vasconcelos, M.Vargas-Vera, D. Bental,

Department of Artificial Intelligence,

University of Edinburgh.

Abstract

Much of the skill of Prolog programming comes from the ability to harness its comparatively simple syntax in sophisticated ways. It is possible to provide an account of part of the activity of Prolog programming in terms of the application of techniques—standard patterns of program development which may be applied to a variety of different programming problems. Numerous researchers have attempted to provide formal definitions of Prolog techniques—for example, (Brna *et al.*, 1991; Kirschenbaum *et al.*, 1989; Gegg-Harrison, 1991)—but there has been little standardization of the approach and the computational use of techniques has been limited to small portions of the programming task. We demonstrate that techniques knowledge can be used to support programming in a wide variety of areas: editing, analysis, tracing, transformation and techniques acquisition. We summarize the main features of systems implemented by the authors for each of these types of activity and set these in the context of previous work, using a standard style of presentation. We claim that a techniques-based system which integrates these features would be worth more than the sum of its parts, since the same techniques knowledge can be shared by the different subsystems.

1. Introduction

The notion of a *programming technique* was first described thoroughly by Brna *et al* (1991). Informally, a programming technique is a common code pattern used by programmers in a systematic way and which is independent of any particular algorithm or problem domain but specific to a particular programming language, in our case Prolog. Brna suggests a number of ways in which having a knowledge of techniques may be useful in both Prolog programming environments and in teaching

Prolog programming skills.

Our work involves developing the notion of Prolog programming techniques. In this paper we describe the progress we have made in applying techniques knowledge to each of the stages of program development. We ultimately envisage an integrated techniques-based Prolog programming environment which allows technique knowledge to be shared across these stages. With such sharing, we believe that the potential return of using techniques is considerably increased. However, this work is not complete. Each of the parts described here has been implemented, but this integration in a single framework is not yet available.

In the next section, we introduce the notion of programming technique in more detail. In subsequent sections we concentrate on particular components of the environment, as listed below:

- In Section 3 we describe two techniques-based Prolog editors.
- In Section 4 we show how techniques help with automatic analysis of programs.
- In Section 5 we show the use of techniques in enhancing a Prolog tracer.
- In Section 6 we describe a program transformation approach making use of techniques knowledge.
- In Section 7 we introduce a tool designed to help manage techniques.
- Finally, in Section 8 we summarize and present our conclusions.

2. Prolog Programming Techniques

Programming folklore gives names to a variety of common constructs occurring in programs. In Prolog, loosely defined terms such as ‘accumulator pair’, ‘difference structure’ and ‘failure driven loop’ are used to describe various aspects of predicates (Sterling & Shapiro, 1986; O’Keefe, 1990). Experts have an understanding of such terms and are capable of applying and adapting these constructs in a wide range of circumstances. However, precise definitions of these terms are not available—they are generally only illustrated by giving examples of predicates containing instances of a construct.

Previous research, notably that by Soloway (1985; 1984), has attempted to formalize expert programming knowledge into a form which can be useful. Soloway used a frame-based representation called plans which was used to understand the semantic mistakes made by novices. A similar attempt at formalizing programming knowledge has been made by Rich and Waters (1987; 1985). They take the view that a program is a composition of standard algorithmic fragments, or clichés. Programmers have a knowledge of these clichés, and, in particular, are capable of combining them in appropriate ways. Each cliché consists of some body of code containing holes which need to be filled with other clichés subject to various constraints. The aim of Rich and Water's work was to develop an editor in which the basic action is to select a cliché and then proceed to fill it in. The editor would ensure that all the necessary constraints are satisfied.

Recent psychological evidence is against the idea that programming plans capture expert programming knowledge in a completely appropriate way (Gilmore, 1990). However, we believe that the major practical problem with these plan based approaches is that, since they attempt to be exhaustive in describing programming constructs, there are too many plans and a complex language is required to describe them. Also, because of this complexity, the algorithms needed to combine plans correctly are computationally expensive.

Whereas plans and clichés are intended as a general representation device for expert's knowledge, techniques attempt to capture only the constructs commonly used in a programming language without including any domain or algorithmic knowledge. That is, a set of techniques for a programming language represents an expert's knowledge of that language, rather than programming and problem solving in general. Note that this set will be a subset of the set of plans, and we expect this set to contain only in the order of tens of techniques instead of the thousands needed in a set of plans[1]

We now give some examples of Prolog techniques. Consider the standard Prolog implementation of 'quick' reverse:

```
rev([], R, R).
rev([H|T], R0, R) :-
    rev(T, [H|R0], R).
```

We can easily decompose this predicate into a part which is performing an exhaustive traversal of a list:

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

and a part which is building a list on the way down through the recursion, and allowing the result

[1] see (Waters, 1985) for an estimate of the number of clichés needed for the Programmer's Apprentice.

to be passed back to the top of the recursion:

```
rev(... R, R).
rev(... R0, R) :-
    rev(... [H|R0], R).
```

This part forms an instance of a common construct known as an accumulator pair (O'Keefe, 1990). The two parts are not completely independent because; for example, they share the variable `H`.

Note that the constructs in these two parts may be used in a wide range of programs, and have little to do with the overall task of the `rev/3` program of reversing a list. Techniques are domain independent. We would not expect to see a technique for averaging a list of numbers, but, as here, we would expect to see a technique for recursing down a data-structure, and for maintaining some accumulator while doing so.

Another example of a technique is a difference list, the classic example of which occurs in the `flatten/3` predicate:

```
flatten(L, F) :-
    flatten(L, [], F).

flatten([], F, F) :- !.
flatten([H|T], F0, F) :- !,
    flatten(H, F1, F),
    flatten(T, F0, F1).
flatten(X, F, [X|F]).
```

A difference list can be used in many situations where two lists constructed separately need to be concatenated together. They are often characterized by the pattern of variable names occurring in the second and third argument positions; in `flatten/3` the 'input' `F0` gets passed to the second recursive call rather than the first. Difference lists work in essentially the same way as accumulator pairs; we return to this point in Section 4.

We view a program as being composed of a number of techniques. A natural next step is to consider in what order techniques may be added during program development. Sterling and his colleagues (Kirschenbaum *et al.*, 1989; Lakhotia, 1989) have defined a methodology based on techniques for constructing logic programs. The result is an approach to the structured development of logic programs, summarized as follows:

- When constructing a program we start out with a basic framework 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. We refer to these as *additions*[2].
- An addition may be applied to a skeleton to obtain an *extension* of the skeleton.
- Extensions may be composed to produce completed programs. This involves a 'clausal-join' algorithm described in (Lakhotia & Sterling, 1987).

To implement this scheme, Sterling makes some commitments about how techniques are represented. Skeletons take the form of simple programs. Additions are mappings between programs adding arguments in a consistent way. For example, their skeleton for traversing a list is:

```
traverse_n([X|Xs]) :-
    case_1(X), traverse_n(Xs).
traverse_n([X|Xs]) :-
    case_2(X), traverse_n(Xs).
    ⋮
traverse_n([X|Xs]) :-
    case_n(X), traverse_n(Xs).
traverse_n([]).
```

This skeleton allows for the general case when the elements of the list need to be distinguished in 'n' different ways. It may be specialized to have only one recursive clause:

```
traverse_1([X|Xs]) :-
    traverse_1(Xs).
traverse_1([]).
```

The result of applying their 'accumulator' addition to this skeleton is:

```
traverse_1_accummulate(Xs, Total) :-
    traverse_1_accummulate(Xs, [], Total).

traverse_1_accummulate([X|Xs], Current, Final) :-
    update(Xs, Ys, Current, Next),
    traverse_1_accummulate(Ys, Next, Final).
traverse_1_accummulate([], Final, Final).
```

This adds in the usual pair of arguments as well as the extra initialization clause. Note that this is still very general; it is necessary to specialize this by providing an appropriate definition of `update/4` which will perhaps need to be modified by adding an extra argument containing `X`.

[2] Sterling uses the term 'technique' but we have renamed it here to avoid confusion.

We have seen that techniques are a formalization of common practice which allow a uniform treatment of standard constructs. Techniques are not designed to be devices for managing complexity, in the way, for example, that a module system is designed. However, a uniform treatment of constructs supports tools which relieve the burden of the details of programming and so addresses the complexity problem indirectly.

3. Techniques in Editing

Editors take a central rôle in programming environments. There has been considerable research into developing editors which are sensitive to the structures of the language being edited (Neal & Szwillus, 1992). These structures often reflected the syntactic block structure of the language, but other approaches are possible. Perhaps the most well known of these is KBEmacs (Rich & Waters, 1987), where editing operations worked over structures known as clichés. We described clichés in Section 2.

Here we describe two editors where the editing structures are techniques. In some ways these editors work in the same manner as KBEmacs, but because the set of techniques is smaller compared with that of clichés, the mechanisms required in the editors are somewhat simpler.

One approach is described by Robertson (1991). This is based directly on Sterling's approach to constructing Prolog programs as illustrated above. He defines skeletons and additions using a simple notation in Prolog. The editor is targeted primarily at novices; for example, the editor sets particular programming problems and incorporates a mechanism to automatically test a student's program against sample calls for these problems. However, the approach could in principle be used in a similar editor for experienced Prolog programmers.

Robertson's editor is highly constrained in that the user may only select a skeleton or addition from a limited choice. The programs constructed are syntactically correct and are guaranteed not to have any arity mismatches. Although the control flow of the program is determined by the initial choice of skeleton, the editor does not check that later additions do not affect that control flow. In this sense the editor is only as good as the set of skeletons and additions it supports.

In a similar way, the applicability of the editor is limited primarily by the library of techniques available to the programmer. With an extensive library the editor provides a structured approach to constructing programs using standard, reusable components.

In our second approach to techniques-based editing, Bowles (1994; 1993) describes an editor named Ted which is also targeted at novices. The aim of Ted is to help novices learn Prolog by providing convenient patterns with which to view programs, and support the process of combining these patterns and learning in what circumstances these patterns are appropriate. This work is motivated by van Someren's suggestions that novices' difficulties with Prolog may be partly alleviated by making common Prolog constructs explicit during teaching (van Someren, 1990; van Someren, 1985).

The set of techniques Bowles defines is small and may be described largely within Prolog, however, the range of Prolog covered is restricted by the following conditions:

1. No mutually recursive predicates.
2. No doubly recursive clauses.
3. Data-structures restricted to lists, atoms and numbers.

Rather than following Sterling's approach to techniques, Bowles defines techniques as common patterns of code which capture the relationship between the head and recursive arguments in the recursive clauses of programs. The most simple of them is called the *same* technique. It is used to pass the same value between the head of a clause and a recursive subgoal in the clause. This technique appears (underlined) in the definition of `rev/3`:

```
rev([], R, R).
rev([H|T], R0, R) :-
    rev(T, [H|R0], R).
```

This approach to techniques is that they capture some relationship between an argument position in the head of a recursive clause and the same argument position in the recursive subgoal. *Same* is the most simple instance of this where the values in these two argument positions are the same; that is, the relationship captured is identity.

Other techniques involve lists. Here the relationship between the head and subgoal arguments is that one argument is a list and the other is the tail of that list. The technique *list head* is used when the head value is the list and the subgoal value is its tail. This appears in the first argument of `rev/3`:

```
rev([], R, R).
rev([H|T], R0, R) :-
    rev(T, [H|R0], R).
```

The technique *list subgoal* is used where the subgoal value is the list and the head value is its tail, as appears in the second argument of `rev/3`:

```
rev([], R, R).
rev([H|T], R0, R) :-
    rev(T, [H|R0], R).
```

In general the relationship between these argument positions is defined by some extra subgoal in the body of a clause. There are two instances of this; where this extra subgoal occurs before the recursive call and where it occurs after it. An example of the latter occurs in the usual implementation of naive reverse list:

```
nrev([], []).
nrev([H|T], R) :-
    nrev(T, TR),
    append(TR, [H], R).
```

This is named the *after* technique:

```
p(X) :-
    p(Y),
    g(Y,X).
```

The value of `X` is constructed in the subgoal `g` from the value `Y` produced by the recursive call.

Corresponding to this is the *before* technique:

```
p(X) :-
    g(X,Y),
    p(Y).
```

This occurs in the following program to compute the length of a list:

```
length([], L, L).
length([H|T], L0, L) :-
    L1 is L0+1,
    length(T, L1, L).
```

Here the subgoal is a call to the builtin `is/2` which is used to increment the counter on the way down through the recursion.

This characterization of techniques leads to a view of clauses as being composed of a number of techniques which share variables. The techniques capture patterns of code within clauses, but they do not attempt to classify the patterns across several clauses. Bowles does however define a number of conditions on valid combinations of techniques across clauses. For example, a predicate such as:

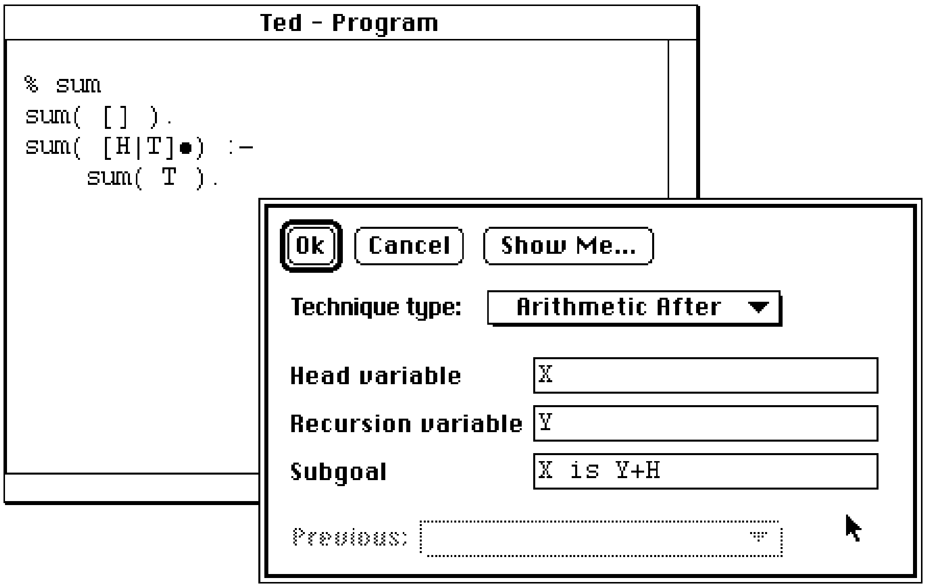

Figure 1: A snapshot of Ted showing the edit dialog for the *after* technique.

```
p([]).
p([H|T]) :- p(T).
p(T) :- p([H|T]).
```

contains a combination of *list head* and *list subgoal* techniques. This may be regarded as invalid since it involves both constructing and deconstructing a data-structure in the same argument position; at the very least this is bad programming style. It is also possible to rule out certain combinations as involving incompatible types.

Ted provides a straightforward syntax editor with a flexible, point-and-click interface supplemented with an edit operation which adds a technique to a recursive clause.

Figures1 shows a snapshot of Ted during the construction of a sum list predicate. Clicking on the program—the place is indicated by the dot on the program display—pops up a dialog in which details of a technique are entered. In this case the technique is an instance of *after* where the subgoal involves arithmetic operation in a call to the Prolog `is/2` built in predicate. Clicking 'Ok' dismisses the dialog and results in the snapshot in Figure 2. To finish the predicate the user will need to add another argument to the first clause.

In these edit dialogs, Ted generates appropriate variable names for some of the fields—n this example, `X` and `Y` are generated. These names are suitable because they do not appear elsewhere in the clause. Ted checks this condition when the dialog is dismissed. This ensures that the technique

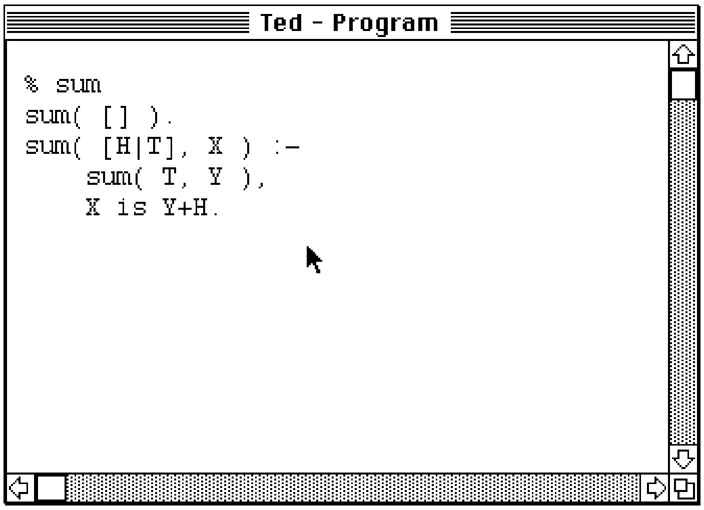

Figure 2: A snapshot of Ted on completion of the *after* technique edit.

does not interact in some unexpected way with the rest of the clause. The details of the subgoal field are entered by the user. This subgoal must involve `X` and `Y`, but can also refer to any other variable in the clause. Here, the variable `H` is used.

The actual edit operations implemented by Ted are very simple. Once the checks on variable names have been performed, Ted has to add an argument to each of the clause head and recursive subgoal, and, in the case of the *before* and *after* techniques, add an extra subgoal.

In both of the editors we have described, the mechanism required to merge techniques into programs is quite simple, although the notion of correctness supported by the editors is not so ambitious as in KBEmacs. However, a major failing of KBEmacs produced by the Programmer's Apprentice project (Rich & Waters, 1987) was the amount of constraint checking necessary to ensure clichés were composed correctly. Whilst this relieves the programmer from a considerable burden, it results in a degradation in the performance of the editor and increases the complexity of cliché definitions. Subsequent work attempted to address these problems (Tan, 1987). Another problem with the use of clichés is that there is potentially a large number of them, and there are access problems associated with trying to find an appropriate cliché in this collection.

In our editors, techniques have been used to factor out common code patterns and allow the users to add them to programs in a convenient way. In Robertson's editor techniques are viewed as standard software components and techniques based editing allows programs to be easily constructed from them. In Ted, with its simple view of how clauses are composed from techniques, novices can focus quickly on the main issues of writing Prolog code such as relationships between arguments and the effects of clause orderings.

The two editors view techniques in different, though compatible, ways. Each view is more appropriate to their circumstances. Ted views techniques at a clause level. This captures common patterns within clauses, but allows students using Ted to play with alternative combinations of techniques across clauses, and so learn about the effects of clause ordering on the behaviour of their programs. Also structures which an expert may regard as being atomic, such as accumulator pairs, are treated by Ted as a composition of techniques. Robertson's editor, on the other hand, is intended to help develop programs reliably and quickly. Thus the techniques used define quite large subparts of the function of the program, though the definition of additions and skeletons is correspondingly complex. These views are compatible. The representation for skeletons and additions could certainly make use of Ted style clause level descriptions of code (suitably generalized to cover a wider range of Prolog). Similarly, as mentioned above within Ted there is a mechanism to highlight incompatible techniques across clauses.

4. Techniques and Automated Program Analysis

In this section we describe how techniques can be used to aid automated program analysis. Automated program analysis allows the programming environment to criticize the programs under development. This is particularly valuable in the field of tutoring where it is necessary to give novices detailed feedback on their programming attempts.

Here we describe two particular applications of such analysis; namely, detecting bugs in novices' code and identifying poor design decisions in working code.

4.1. Identifying Buggy Programs

All methods to identify bugs in student programs follow a similar approach (Johnson, 1985; Murray, 1988; Looi, 1988). These debuggers contain some description of the alternative ways of implementing each programming task the student has been set. These ideal solutions are stored in a canonical form which ignores as much as possible irrelevant details of the implementation—for example, the names of identifiers used in the program. Debugging is reduced to matching the student's solution against these descriptions. The description which best matches the student's solution is assumed to be the implementation the student intended, and any mismatches between the description and the student's solution are regarded as bugs which may be corrected.

This approach assumes that there are available analysis procedures which are capable of parsing the student's solution into the canonical description language (or equivalently performing a nonsyntactic match of the solution and a description). For example, Murray (1988) uses the Boyer-Moore system to prove the equivalence between parts of a student's solution and the descriptions. Looi makes use of a range of different analyzers which infer features used in the descriptions—for example, the type of data-structures.

The matching process is approximate since there is a wide variation in the forms programs may take. The problem is to describe canonical solutions in a way which is both accurate, sufficiently flexible to allow all sensible variations on the solution and in which it is not too difficult to add more canonical descriptions. Work by Bowles (1991) has shown that programming techniques form a useful intermediate language for these descriptions. That is, a predicate is described mainly by using a set of techniques.

Consider the following two versions of the standard problem to flatten nested lists. The first, using difference lists, is the most familiar:

```
flatten_dl(L, F) :-
    flatten_dl(L, [], F).

flatten_dl([], F, F) :- !.
flatten_dl([H|T], F0, F) :- !,
    flatten_dl(H, F1, F),
    flatten_dl(T, F0, F1).
flatten_dl(X, F, [X|F]).
```

The second uses an accumulator:

```
flatten_ac(L, F) :-
    flatten_ac(L, [], F).

flatten_ac([], F, F) :- !.
flatten_ac([H|T], F0, F) :- !,
    flatten_ac(T, F0, F1),
    flatten_ac(H, F1, F).
flatten_ac(X, F, [X|F]).
```

Although difference lists and accumulators are traditionally regarded as quite distinct, the only difference between these two programs is the order of the two subgoals in the recursive case. It is difficult for a simple syntax-based analyzer to distinguish between these two predicates. This problem is compounded when buggy versions of the predicates need to be distinguished.

This problem is important because if the inappropriate 'ideal' solution is chosen as the best target match against a student's buggy predicate, then the bugs the analyzer produces may be described in a confusing way for the student. In this case it is probably best to avoid explicitly mentioning difference lists or accumulators, and to use more general descriptions.

To address these problems Bowles defined techniques using data flow, as well as syntactic, information. Both of the above predicates have similar data-flow between the variables involved, despite being syntactically different. The data-flow may be described by sets of pairs of a data-flow relationship $X \ll Y$, which may be informally read as `X` contains `Y`. The data-flow in both the above programs may be described $\{F \ll F1, F1 \ll F0\}$. Difference lists and accumulators may be distinguished by annotation of sample code with these pairs:

```
dl(F0, F) :-
    dl(F1, F),   F≪F1
    dl(F0, F1).  F1≪F0

ac(F0, F) :-
    ac(F0, F1),  F1≪F0
    ac(F1, F).   F≪F1
```

Thus if the debugger detects dataflow patterns which are similar to these, then this maybe taken as evidence of the use of a, possibly buggy, instance of an accumulator pair or a difference list. Under this assumption, other parts of the predicate may be analyzed separately.

This representation of techniques allows the analyzer to focus first on the data-flow of the predicates before dealing with syntax. Explicitly using techniques to describe ideal solutions allows the important features to be detected more easily and reduces the search involved in matching predicates against canonical descriptions.

4.2. Criticizing Design Decisions

Novices who are learning to program in Prolog must learn to choose and implement the right Prolog programming techniques. The choice of an appropriate programming technique depends both on the task that the student intends to perform and also on the data structures that are being manipulated. Programming exercises for beginners often restrict the students' range by giving specifications for small tasks which specify the data structures (for example, reverse a list), and may be intended to exercise a particular subset of programming techniques. As novices move on to more advanced (and more realistic!) programming exercises they are required to make more of these decisions for themselves. Novices may have difficulty in mapping the task onto the appropriate techniques, in choosing good data representations and in combining the right programming technique with the data representation. Such decisions are difficult to detect because they may not exhibit obviously incorrect behaviour. Nethertheless, they ought to be detected and highlighted by a tutoring system concerned with good programming practice.

Bental (1994) has addressed the problem of recognizing design decisions in a student's completed Prolog program. She has distinguished between decisions that are expressed in terms of the problem that has been set and decisions that are made in terms of the programming language. The latter are viewed in terms of which programming techniques the student has chosen to use. Bental (1993) has also distinguished between decisions about data representations and about how those representations should be manipulated.

As an example consider the following predicate whose intended task is to search for an empty square in the board of a noughts-and-crosses (tic-tac-toe) program, and return the position of the empty square:

```
free_square(Board, Position) :-
   free_square(Board, [1,2,3,4,5,6,7,8,9], Position).

free_square([H|_], [Position|_], Position) :-
    empty_square(H).
free_square([_|T], [_|T1], Position) :-
   free_square(T, T1, Position).
```

This program does fulfill the task requirements for a noughts-and-crosses board of nine squares. The problem is that the student has explicitly written a list of board positions—that is, numbers from 1 to 9—into the predicate, and traversed down this list whilst traversing the list representing the board. A better designed implementation would have used a simple incremental counting technique instead of the list recursion. A counter would be more standard, more robust (what happens if a larger board were used?) and less prone to typing errors (such as leaving out one of the numbers). This error might indicate that the student does not really understand Prolog arithmetic techniques, and certainly a tutoring systems ought to highlight this alternative implementation.

Correct solutions to the noughts-and-crosses problem are stored in a task hierarchy where the lowest level of subtask are represented as a set of skeletons and additions. Matching these solutions against a student's code begins by taking an analysis-by-synthesis approach (Reiser *et al.*, 1985) in which she applies additions to skeletons and matches the resulting code against the student's code. Sterling's clausal join algorithm (Lakhotia & Sterling, 1987) is used on the system's generated implementations to generate further composed implementations that can be matched against the student's code. A purely analysis-by-synthesis approach does not work well if one of the components of the student's composed procedure cannot be recognized. In this case, Bental's analysis works backwards from the student's composed procedure to its components, to identify the skeletons and techniques in those components for which this is possible and to isolate the other components. To do this, Bental (1992) defines an algorithm called clausal split which acts as an inverse of clausal join.

In this way, Bental's critiquing system is able to infer that the board is represented as a list of lines, and that the context in which the predicate is called suggests that the intended task is to find an empty square but that the implementation does not conform to any known implementation of this task. The system then uses clausal split to separately identify the recursion on the board and the code that should implement a counter. The recursion on the board is correct and so the error is isolated as being within the counting technique.

Thus Bental uses techniques as a representation device for the good solutions, and clausal join and split are used to isolate poor choices of programming techniques.

5. Techniques and Program Tracing

In the field of explanation for knowledge-based systems it is commonly argued that traditional execution traces, whilst perhaps adequate to explain what a program does, are inadequate to justify to users that such behaviour is reasonable. Systems such as XPLAIN (Swartout, 1983) attempt to remedy this deficiency by constructing the initial program within an environment which forces the system designer to add justifications of design choices when developing the program. This extra information may then be used to justify, in terms of the programmer's design plan, why a particular behaviour is observed in tracing.

An analogous problem is experienced when teaching elementary Prolog. Here the tracer is often used as an aid to understanding why a predicate has been defined in a particular way. Unfortunately, a bare execution trace (even highly sophisticated ones such as (Eisenstadt & Brayshaw, 1988)) cannot explain to novices what parts of the program are pivotal in obtaining particular behaviours. To supply this extra information it is necessary for the tracer to employ information about the way in which the program has been built. Since a techniques editor is capable of providing a limited record of the stages of construction of the program, by keeping track of the way in which techniques are applied in the editor, it is possible to enhance explanations of a program's behaviour by using this additional information. A tracing system based on this approach is described in (Gabriel, 1992). The basic approach will be described using the standard `append/3` definition:

```
append([], X, X).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Suppose that we require an explanation of the way in which this program can be used to satisfy the goal:

```
| ?- append([a,b], [c,d], X).
```

Gabriel's tracer can be used in a conventional manner - for example to follow the trail of goals through call and exit ports:

```
1  Call:  append([a,b], [c,d], X) ?n
1.1  Call:  append([b], [c,d], V1) ?n
1.1.1  Call: append([], [c,d], V2) ?n
1.1.1  Exit: append([], [c,d], [c,d]) ?n
1.1  Exit:  append([b], [c,d], [b,c,d]) ?n
1  Exit:  append([a,b], [c,d], [a,b,c,d]) ?n
```

However, the tracer also has access to the following information about the construction history of `append/3`, which can be supplied by a techniques editor similar to those described in Section 3. A sketch of the steps of construction, with their associated information, is as follows:

- The flow of control of the program is defined using a traversal skeleton. The first argument is used, in the mode of the given top-level goal, to deconstruct a list while recursing on its tail.
- We then apply a context technique. The second argument is used to carry a list through the recursion.
- We then apply a construct list technique. The third argument is used, in the mode of the given top-level goal, to construct a list while recursing on its tail.

- We then identify variables which are shared between skeletons/techniques:
  - The variables in the second and third arguments of the base case are identified with one another.
  - In the recursive clause, the head of the list in the first argument is identified with the head of the list in the third argument.

Using this extra information, the tracer can provide detailed explanations relating the program construction operations to the programs behaviour and can also give higher level indications of the flow of control in the program. An example of the former type of explanation is given by the output shown below, which explains how the goal `append([], [c,d], V2)` was satisfied:

```
BASE CASE CALL:
Tests that the first argument (i.e. []) is an empty list.

the first argument (i.e. []) is the controlling argument having been
fully unpacked.

the third argument (i.e. V2)
is passed the value of:
the second argument (i.e. [c,d])
```

Notice that this gives us more information than the original trace. In particular it takes into account that the first argument is providing the flow of control and that the second and third arguments have been identified with one another as a means of instantiating the variable `V2`.

Since the explanations produced by the tracer are driven by generating "pseudo-English" text from the techniques information it is possible to obtain explanations of this sort at any exit port of the tracer for any program containing the required techniques information. For example, it can generate a techniques-based explanation of how the goal `append([a,b], [c,d], X)` was satisfied:

```
the third argument (i.e. X)

is passed the value of:

the value obtained by adding:
    the head of the list given by:
        the first argument (i.e. [a,b])
    with value: a
to the front of the list given by:
    the third argument from the deeper call (i.e. [b,c,d]).
giving the result: [a,b,c,d]
```

In addition to adding detail to the explanation of the program it is also possible to provide more general explanations of control of flow. This can be obtained because the skeletons and techniques give a "directionality" to the arguments of the predicate. The tracer gives an indication of this by displaying any path in the proof tree as a sequence of subgoals and, based on the techniques information, placing arrows on the sequence to represent data flow between arguments. For our running example, the display is as follows:

$$\mathrm{append}([a,b],\ [c,d],\ X/[a,b,c,d])$$

$$\mathrm{append}([b],\ [c,d],\ V1/[b,c,d])$$

$$\mathrm{append}([],\ [c,d],\ V2/[c,d])$$

6. Techniques and Program Transformation

Source level program transformation allows clear but inefficient programs to be translated into more efficient equivalents. Burstall and Darlington (1977) defined fold/unfold steps for transforming functional programs, and Tamaki and Sato (1983) reconstructed this work for logic programming. In this section we show that having a techniques description of a program provides convenient support for program transformation.

The following is a simple application of program transformation. Occasionally, a program requires two separate properties to be computed for a particular data-structure. The most straightforward approach to this is to compute these properties separately, but this leads to the inefficiency of processing the data-structure twice. A small example of this occurs in the following program to compute the average of the elements in a list:

```
average(List, Average) :-
    count(List, Count),
    sum(List, Sum),
    Average is Sum/Count.

count([], 0).
count([H|T], Count) :- count(T, C1), Count is C1+1.

sum([], 0).
sum([H|T], Sum) :- sum(T, S1), Sum is S1+H.
```

Sterling's clausal-join algorithm allows these side-effect free programs to be merged (Lakhotia & Sterling, 1987; Sterling & Lakhotia, 1988). This makes use of following *join specification*:

```
count_sum(List, Count, Sum) :-
    count(List, Count),
    sum(List, Sum).
```

This is first expanded by unfolding and tidying the definitions of the `count/2` and `sum/2` programs:

```
count_sum([], 0, 0).
count_sum([H|T], Count, Sum) :-
    count(T, C1),
    Count is C1+1,
    sum(T, S1),
    Sum is S1+H.
```

and then folded back using the join specification itself to replace the unwanted recursive calls:

```
count_sum([], 0, 0).
count_sum([H|T], Count, Sum) :-
    count_sum(T, C1, S1),
    Count is C1+1,
    Sum is S1+H.
```

This approach is most effective if the two merged programs share the same flow of control over the data-structure. Sterling assumes that the two programs were derived from the same skeleton, which guarantees that the control flow is the same. In the above case, both `count/2` and `sum/2` are based on the following skeleton to recurse down a list:

```
s([]).
s([H|T]) :-
    s(T).
```

This is an instance of using a technique description of a program to suggest an effective transformation.

Recently, Vargas-Vera (1994) has applied this idea to extend the range of transformations possible. Essentially, the same approach is used: definitions are unfolded out, tidied up and folded back together. Consider the following example:

```
get_even([], []).
get_even([H|T], [H|E]) :- even(H), get_even(T, E).
get_even([H|T], E) :- odd(H), get_even(T, E).

sum_even(L, S) :- get_even(L, Es), sum(Es, S).
```

A description of the output argument of `get_even/2` shows that it is building up a list, whereas the input of `sum/2` is decomposing the same list. This suggests that it is possible to compose these two calls. The first step is to unfold the definition of `get_even/2` in `sum_even/2`:

```
sum_even([], S) :-
    sum([], S).
sum_even([H|T], S) :-
    even(H), get_even(T, Es), sum([H|Es], S).
sum_even([H|T], S) :-
    odd(H), get_even(T, Es), sum(Es, S).
```

Then unfold the calls to `sum/2` in the first and second clause:

```
sum_even([], 0).
sum_even([H|T], S) :-
    even(H), get_even(T, Es), sum(Es, SEs), S is H+SEs.
sum_even([H|T], S) :-
    odd(H), get_even(T, Es), sum(Es, S).
```

Finally, fold back the original definition of `sum_even/2`:

```
sum_even([], 0).
sum_even([H|T], S) :-
    even(H), sum_even(T, SEs), S is H+SEs.
sum_even([H|T], S) :-
    odd(H), sum_even(T, S).
```

This program avoids building and then decomposing the intermediate list.

Of course this approach is only possible if technique descriptions of programs are conveniently available. Rather than attempting to infer these descriptions for the program text, Vargas-Vera assumes that the programs were developed using a techniques editor of the kind described in Section 3, and which records the history of program development.

7. Managing Techniques

We have shown that techniques can be applied at various stages of program development but, until now, have not considered the problem of acquiring techniques knowledge. Although it seems that a small number of general techniques can be used to construct a wide range of Prolog programs, it is not clear that all users will find these general definitions easy and efficient to use. A more likely scenario is that particular user groups will require techniques customized to their domain of application. For example, someone interested in building an expert system might prefer to have the skeletal structure of a rule interpreter available as a single technique, rather than having to construct it from primitive components.

We believe an additional tool is needed to manage techniques, though as we demonstrate here such a tool can support other useful facilities. Consider the problem of developing skeletons for describing control flow of programs as used by Sterling. Whilst there are a number of standard traversals of data-structures—for example, depth-first, left-to-right searches—skeletons are also dependent on the data-structures being manipulated in a program. We believe there is a need for a tool to construct new skeletons, and other techniques, appropriate to new data-structures, and also ensure various useful properties of these techniques. For example, we might expect skeletons to always terminate.

In this section, we describe an initial design for a *Techniques Meta Editor*, or TME which addresses these issues (Vasconcelos, 1992; Vasconcelos, 1993). Such a tool provides an interface for expert programmers wishing to design and test techniques, offering commands allowing the insertion, removal and alteration of components of a library of techniques. It also helps programmers modify existing techniques and define relationships between them.

TME defines a 'most general' technique, and allows the user to specialize it in a number of orthogonal dimensions. This is in some ways similar to Bundy's recursion editor (Bundy *et al.*, 1991). TME considers a technique as a sequence of clauses of the form:

$$\begin{array}{ll} \texttt{t(X) :-} & \langle \textit{required}(\bar{\texttt{Y}}) \rangle \\ \quad \texttt{process(X,}\bar{\texttt{Y}}\texttt{,}\bar{\texttt{Z}}\texttt{,}\bar{\texttt{R}}\texttt{),} & \langle \textit{offer}(\bar{\texttt{Z}}) \rangle \\ \quad \texttt{t(R}_1\texttt{), } \ldots \texttt{, t(R}_n\texttt{).} & \forall\, \texttt{R}_i \in \bar{\texttt{R}} \end{array}$$

where $\bar{\texttt{Y}}$, $\bar{\texttt{Z}}$ and $\bar{\texttt{R}}$ are (possibly empty) vectors of variable symbols. Here the relationship between a single argument position and a number (possibly zero) of recursive calls is defined by a subgoal `process`. This subgoal may be expanded to any number of system or user defined subgoals which may act as test or may be used to extract or insert values. The annotation *required* acts as a marker for variables whose values must be obtained from elsewhere in the clause and the annotation *offer* acts as a marker for variables whose values are available for use elsewhere in the clause.

TME guides its users through a series of interactions in which a generic template embodying the definition above is gradually customized, by specifying the number of clauses and the number of recursive (vector $\bar{R}$) and non-recursive (vectors $\bar{Y}$ and $\bar{Z}$) values of each clause, by renaming variables, by expanding the subgoal `process` and by reordering the subgoals in the clause. For example, the user may initially restrict the template to two clauses, one non-recursive with no required or offered values, and the other with one recursive call:

```
t(X) :-
    process(X).
t(X) :-                 ⟨required(Ȳ)⟩
    process(X,Ȳ,Z̄,R),   ⟨offer(Z̄)⟩
    t(R).
```

Subsequently, the user may specialize this to recurse over lists, offering the head of the list in each recursive step and requiring no other values:

```
list(X) :-
    X = [].
list(X) :-
    X = [Z|R],  ⟨offer(Z)⟩
    list(R).
```

We might define skeletons to be techniques such as this which have no required values. An alternative specialization would be to develop a technique to maintain a sum:

```
sum(X) :-
    X = Z.      ⟨offer(Z)⟩
sum(X) :-       ⟨required(Y)⟩
    Z is X+Y,   ⟨offer(Z)⟩
    R = Z,
    sum(R).
```

Here the user has replaced `process` with appropriate base and recursive calls, and renamed the entire technique. Similarly, the user could also define an 'output' technique:

```
out(X) :-       ⟨required(Y)⟩
    X = Y.
out(X) :-
    X = R,
    out(R).
```

A further operation supported by TME is composing together techniques. For example, the `sum` and `out` techniques above may be composed to give an accumulator pair to calculate and return the sum:

```
sum_out(A, B) :-
    A = B.              ⟨offer(A)⟩
sum_out(A, B) :-        ⟨required(Y)⟩
    A1 is A+Y,          ⟨offer(A1)⟩
    B1 = B,
    sum_out(A1 B1).
```

This composition is valid since the clauses and recursive calls match and also the value required in the base case of the `out` technique is offered by the `sum` technique. TME would also perform transformations to tidy up these techniques by removing superfluous calls to `=/2`.

A value `Y` is still required by the `sum_out` technique. This must be provided by further compositions with other techniques to yield a completed program. The `list` technique above is a suitable candidate for this, but note that any technique recursing over some linear data-structure and providing some value at each recursive step is also a suitable candidate.

In this way TME supports the partial definition of techniques in a flexible way, say, by leaving the number of recursive calls or the type of the data unspecified. This maximizes the possibilities of composing techniques and ensures a parsimonious representation for the set of techniques provided in the environment. We can imagine the set of techniques developed by TME to be organized into hierarchies of partial technique definitions each representing a set of specializations. Any technique which may compose with a partial definition will also compose with its specializations. This mechanism supports the reuse of technique definitions, but it is also true that any properties of a partial definition may also be inherited by its specializations. Examples of such properties include termination conditions, complexity values and possible transformations or compilation optimizations.

8. Summary

In this paper we have shown that techniques, capturing small commonly occurring patterns of code, may be effectively used in a wide variety of the subtasks of program development. We have demonstrated that comparatively modest amounts of techniques information can be harnessed to provide significant improvements in the level of support provided to programmers in major phases of program construction: editing (Section 3), analysis (Section 4), tracing (Section 5), transformation (Section 6) and techniques acquisition (Section 7). There are other areas of application which have not been addressed. For example, in teaching techniques provide a useful Prolog syllabus which may be used to order the presentation of examples during a teaching session (Gegg-Harrison, 1991).

Each of the parts we have described has been implemented, but the goal of an integrated Prolog programming environment supported by techniques knowledge has yet to be attained. Although we have demonstrated the utility of the individual components of a techniques-based Prolog environment the task of combining these subsystems is non-trivial, since it requires a uniform account of the use of techniques which is both appealing to users and representationally powerful. In a collaborative project, we are conducting empirical tests of the use of the Ted editor as part of Prolog programming courses and hope that this will inform our attempts to refine and integrate our existing tools.

Acknowledgements

A. Bowles and D. Robertson are supported by the SERC/ESRC/MRC Joint Council Initiative in Cognitive Science/HCI, Grant number G 9030396. W. Vasconcelos is on leave from State University of Ceará, Ceará, Brazil and is sponsored by the Brazilian National Research Council (CNPq), Grant number 201340/91-7. M. Vargas-Vera is supported by the National University of Mexico. D. Bental was supported by a SERC studentship.

References

BENTAL, D. 1992. Using Clausal Join and Clausal Split to Recognise Language-Specific Programming Design Decisions. *Pages 37–40 of: AAAI Conference Workshop on AI and Automated Program Understanding.*

BENTAL, D. 1993. Detecting Design Decisions About Data Structures in Prolog Programs. *Pages 217–224 of:* BRNA, P., OHLSSON, S., & PAIN, H. (eds), *World Conference on Artificial Intelligence in Education.* AACE.

BENTAL, D. 1994. *Recognising the Design Decisions in Prolog Programs as a Prelude to Critiquing.* Ph.D. thesis, Department of Artificial Intelligence, University of Edinburgh.

BOWLES, A. 1991. *Detecting Prolog Programming Techniques Using Abstract Interpretation.* Ph.D. thesis, University of Edinburgh.

BOWLES, A. 1994. *A Techniques Editor for Prolog Novices.* Internal software report, available from author.

BOWLES, A., & BRNA, P. 1993. Programming Plans and Programming Techniques. *Pages 378–385 of:* BRNA, P., OHLSSON, S., & PAIN, H. (eds), *World Conference on Artificial Intelligence in Education.* AACE.

Brna, P., Bundy, A., Dodd, T., Eisenstadt, M., Looi, C.K., Pain, H., Robertson, D., Smith, B., & van Someren, M. 1991. Prolog Programming Techniques. *Instructional Science*, **20**(2/3), 111–134.

Bundy, A., Grosse, G., & Brna, P. 1991. A Recursive Techniques Editor for Prolog. *Instructional Science*, **20**(2/3), 135–172.

Burstall, R.M., & Darlington, J. 1977. A Transformation System for Developing Recursive Programs. *Journal ACM*, **24**(1), 44–67.

Eisenstadt, M., & Brayshaw, M. 1988. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, **5**(4), 277–342.

Gabriel, D.S. 1992. *TX: A Prolog Explanation System.* M.Phil. thesis, Department of Artificial Intelligence, University of Edinburgh.

Gegg-Harrison, T.S. 1991. Learning Prolog in a Schema-Based Environment. *Instructional Science*, **20**(2/3), 173–192.

Gilmore, D.J. 1990. Expert Programming Knowledge: A Strategic Approach. *Chap. 3.2, pages 223–234 of:* Hoc, J-M., Green, T.R.G., Gilmore, D.J., & Samurcay, R. (eds), *The Psychology of Programming.* London: Academic Press.

Johnson, W. L. 1985. *Intention-Based Diagnosis of Errors in Novice Programs.* Ph.D. thesis, Yale University.

Kirschenbaum, M., Lakhotia, A., & Sterling, L. 1989. *Skeletons and Techniques for Prolog Programming.* Technical Report 89-170. Case Western Reserve University.

Lakhotia, A. 1989. Incorporating Programming Techniques into Prolog Programs. *Pages 426–440 of:* Lusk, E., & Overbeek, R. (eds), *Proceedings of the North American Conference on Logic Programming*, vol. 1. MIT Press.

Lakhotia, A., & Sterling, L. 1987. *Composing Logic Programs with Clausal Join.* TR 87-25. Computer Engineering and Science Department, Case Western Reserve University.

Looi, C. K. 1988. *Automatic Program Analysis in a Prolog Intelligent Teaching System.* Ph.D. thesis, University of Edinburgh.

Murray, W. R. 1988. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan Kaufmann.

Neal, L., & Szwillus, G. 1992. Structure-based editors and environments. *International Journal of Man-Machine Studies*, **37**, 395–397. Introduction to special issue.

O'Keefe, R. 1990. *The Craft of Prolog.* MIT Press.

REISER, B., ANDERSON, J., & FARRELL, R. 1985. Dynamic student modeling in an intelligent tutor for Lisp programming. *Pages 8–14 of: Proceedings of the International Joint Conference on Artificial Intelligence.*

RICH, C., & WATERS, R.C. 1987. *The Programmer's Apprentice: A Program Design Scenario.* AI Memo 933A. Artificial Intelligence Laboratory, MIT.

ROBERTSON, D. 1991. A Simple Prolog Techniques Editor for Novice Users. *Pages 190–205 of:* WIGGINS, G.A., MELLISH, C., & DUNCAN, T. (eds), *3rd UK Annual Conference on Logic Programming.* Berlin: Springer-Verlag.

SOLOWAY, E. 1985. From problems to programs via plans: the content and structure of knowledge for introductory Lisp programming. *Journal of Educational Computing Research*, **1**, 157–172.

SOLOWAY, E., & EHRLICH, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions of Software Engineering*, **10**(5), 595–609.

STERLING, L., & LAKHOTIA, A. 1988. Composing Prolog Meta-Interpreters. *Pages 386–403 of:* KOWALSKI, R.A., & BOWEN, K.A. (eds), *Logic Programming: Proceedings of the Fifth International Conference and Symposium.* Cambridge MA: MIT Press.

STERLING, L., & SHAPIRO, E. 1986. *The Art of Prolog.* MIT Press.

SWARTOUT, W.R. 1983. XPLAIN: A System for Creating and Explaining Expert Consulting Systems. *Artificial Intelligence*, **21**(3), 285–325.

TAMAKI, H., & SATO, T. 1983. *A Transformation System for Logic Programs Which Preserves Equivalence.* TR 83-18. ICOT Research Center.

TAN, Y. 1987. *Ace: A Cliché-Based Program Structure Editor.* working paper MIT/AI/WP-294. MIT Artificial Intelligence Laboratory.

VAN SOMEREN, M. W. 1985. *Beginners Problems in Learning Prolog.* Memorandum 54. Department of Experimental Psychology, University of Amsterdam.

VAN SOMEREN, M. W. 1990. What's Wrong? Understanding Beginners' Problems with Prolog. *Instructional Science*, **19**(4/5), 257–282.

VARGAS-VERA, M. 1994. *Guidance during program composition in a Prolog Techniques Editor.* Ph.D. thesis, Department of Artificial Intelligence, Edinburgh University. In preparation.

VASCONCELOS, W. W. 1992. Formalising the Knowledge of a Prolog Techniques Editor. *In: 9th Brazilian Symposium on Artificial Intelligence.*

Vasconcelos, W. W. 1993. Designing Prolog Programming Techniques. *Pages 85–99 of:* Deville, Y. (ed), *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation.* Springer Verlag.

Waters, R. 1985. *KBEmacs: a Step Toward the Programmer's Apprentice.* Technical Report 753. MIT Artificial Intelligence Laboratory.