Combining Prolog Programs in a Techniques Editing System

†María Vargas-Vera, †Dave Robertson and ‡Robert Inder

†Department of Artificial Intelligence
University of Edinburgh,
80 South Bridge,
Edinburgh EH1 1HN.
Scotland, Great Britain
email: {mariav,dr}@aisb.ed.ac.uk

‡Human Comunication Research Center,
University of Edinburgh,
2 Buccleuch Place
Edinburgh EH8 9CW
Scotland, Great Britain
email:robert@cogsci.ed.ac.uk

Abstract

Techniques editing, as proposed by Sterling et al., allows Prolog programs to be constructed by initially selecting a 'skeleton' which determines the flow of control of the program, and then adding on top of this the extra features required by the program. This means that it is easy to obtain as an end-result of techniques editing not only the final program but also a history of its development, in terms of the skeleton and extensions used to build it. We describe how this program history information can be used to produce efficient combined programs from pairs of initial programs constructed independently by a techniques editor.

Keywords: techniques editor, classification of Prolog programs, join specification, composition methods, unfold/fold transformations, meta-folding operation, program history.

1 Introduction

This paper addresses the combination of two programs constructed by means of a specialised techniques editor, briefly described in Section 2. Currently the techniques editor allows programs to be built by applying a variety of techniques [KLS89] to skeletons, depicting flow of control, and producing enhancements of them. Skeletons themselves correspond to the sort of information that we require in order to provide efficient transformations when combining programs.

To enable the most appropriate choice of combination method the set of programs that we can write in Prolog is divided into classes according to stipulated structural features. The features taken into consideration are: the skeleton employed, number of clauses, data structures and pattern matching used to compose or decompose the data structure. Five classes of programs with the same flow of control were identified: *traverse*, *short_traverse*, *search*, *meta-interpreters* and the *counters* class. We have also five classes with different flow of control. These are the mutants of each of the classes defined before. A program P_2 is a mutant developed from another program P_1 if P_2 is a version of P_1 which has been adapted by adding subgoals which may conditionally terminate a clause or by adding new clauses which process conditions not checked by the initial program P_1.

The contribution of this paper is that a simple *program history* (consisting of a record of the enhancements made to the basic program skeleton when using the techniques editor) can be used in program transformation, reducing user interaction. This program history can be obtained from a techniques editor. An example of how the transformation of the program can be performed easily without introducing several new transient join specifications is described in Section 5.

This paper is structured as follows: Section 2 describes the methodology of stepwise refinement using skeletons and techniques. Section 3 gives a program construction system architecture describing each of its components. Section 4 gives an overview of our composition methods and introduces two methods for combining Prolog programs (the *meta-composition* and *mutant method*). Section 5 presents an example firstly using standard transformation operations and secondly it gives the same example using one of our composition methods (the *mutant* method) which requires knowledge about the program development. Finally Section 6 draws some conclusions; points out the limitations of our system; and gives some ideas for future research.

2 Stepwise Enhancement using Skeletons and Techniques

Kirschenbaum et al. proposed a methodology for construction of Prolog programs called 'stepwise enhancement' [KLS89]. This methodology suggests the idea of developing a program by finding the suitable basic flow of control (*'skeleton'*); once the skeleton has been determined, extra computations are included by applying appropriate *techniques* to produce an *extension.* Separate extensions can be combined to produce the desired program. The extensions can be regarded as another skeleton, permitting us to repeat the process until the final program has been completed.

An extension is a program derived from a skeleton by applying the following modifications repeatedly: renaming the predicate, adding arguments, adding subgoals, reordering arguments and adding clauses which do not change the flow of control. These modifications can be achieved by applying *techniques.* Techniques are defined as standard Prolog programming practices [KLS89]. For instance, a technique allows variables to be carried to the recursive subgoals in a recursive predicate, another technique allows a data structure to be built by traversing the flow of control of the skeleton and by adding appropriate elements to the new data structure. A description of a set of techniques is also given in [KLS89].

This paper does not focus on the techniques editor, though we have implemented two versions of this part of the system. Robertson implemented a simple version of the techniques editor which is based on Sterling's notion of skeletons and techniques [Rob91]. Robertson defines skeletons and techniques using a simple DCG-like notation in Prolog. Currently Robertson's editor is restricted to a small set of skeletons and techniques from which the user can select in order to achieve his/her plan for the program. The second editor, named Ted, was implemented by Bowles [Bow93] and is also targeted at novices. Ted helps novices to learn Prolog by providing convenient patterns with which to construct programs, and allows the process of combining these patterns and learning in what circumstances they are appropriate. This editor uses a different approach to techniques than Sterling. In Bowles' approach techniques, for example, are local to clauses rather than applying across whole predicates. Ted provides a syntax editor consisting of a point-and-click interface supplemented with a set of edit operations which allow a technique to be applied to a clause. Both these editors allow techniques to be merged into programs in a simple way. In Robertson's editor techniques are considered as software components which can be incorporated into a general control flow defined by skeletons, whilst in Ted techniques are viewed at a clause level.

3 A Program Construction System

Our proposed program construction system consists of a Prolog *techniques editor* and a *composition system*, shown in Figure 1. The Prolog techniques editor uses a knowledge base of techniques and skeletons and produces a *program history* and program as output. This information is passed as input to the *composition system* which, by using a transformation rules library and the skeletons knowledge base, produces as output the combined program and a new history for the combined program. The combined program and the new history are recorded for future stages in the composition process.

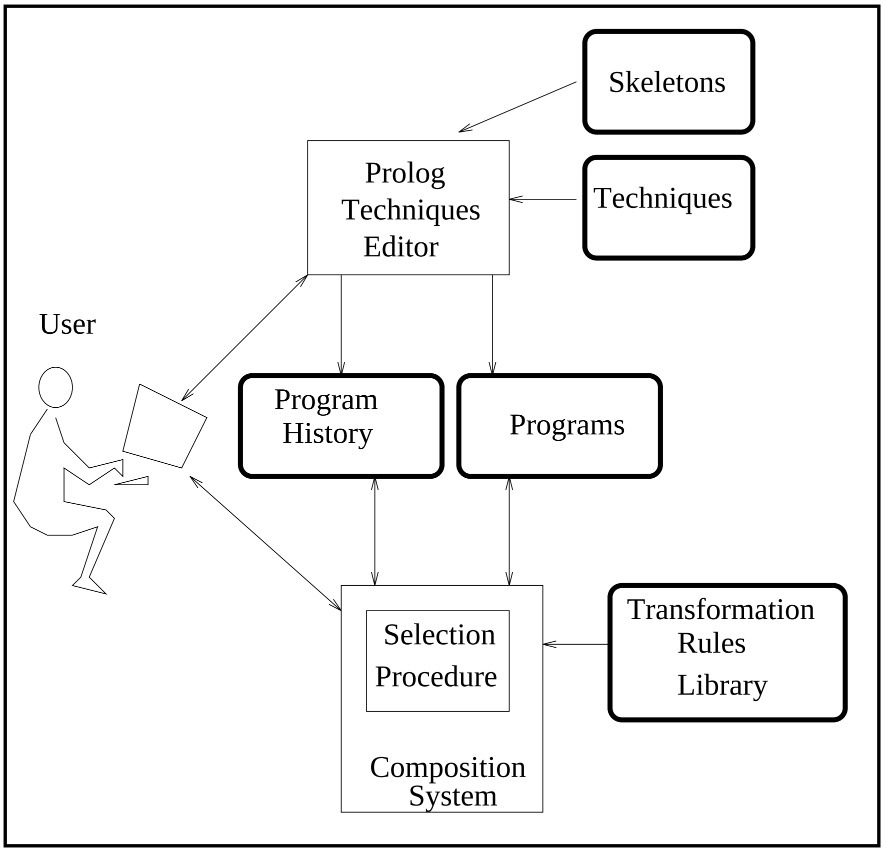

Figure 1: A Program Construction System

The primary component of the composition system is the *selection procedure*. The main function of this is to determine the combining method taking into consideration the initial programs. The input for this *selection procedure* is the program history, which contains information concerning to the skeleton and techniques used in the construction of programs to be combined.

The proposed library of transformations holds only those transformations which preserve the equivalence of programs. Therefore the transformation process does not alter the intended behaviour of the initial programs. This transformation rules library is described in Section 3.3.

3.1 Selection procedure

The selection stage is the kernel of our composition system, since it determines which sequence of transformations will be used. Several approaches can be used at this stage.

- A static program analysis can be performed to determine which skeleton or flow of control was used in each program. The information that the program analysis will provide is basically the skeleton and the set of techniques applied in the construction of the program. Unfortunately it is not always the case that it is possible to determine which skeleton was used, for instance, the skeleton used in a mutant program cannot be easily determined.

- A second approach is proposed by Fuchs [FF91]. He proposes making a program abstraction in order to match the program with a schema defined in Gegg-Harrison's hierarchy of Prolog schemata [GH91]. At the top of this hierarchy are fourteen basic-level schemata which capture simple recursive list processing Prolog programs. The combination of programs is performed by transforming two input program schemata (associated with each program to be combined) into output schema (the resulting combined program). This process is done in three stages: abstraction of the programs to a program schema, selection of a transformation schema with this schema as input and a suitable schema as output and finally specialization of the output schema to the transformed program. User interaction is required during the selection stage. This selection of the output schema is not an easy task for beginners of Prolog because it requires that they know in advance the form of the clauses for the combined program (in particular the form of the induction parameter for each clause).

- A third approach which we adopt is based on the program history. This information reduces the user interaction and, resulting, produces an efficient combined program.

The main decisions in our composition system rely on the program history and do not require a significant amount of user interaction. The questions required to be answered by the user are the name and the arity of the top level predicates which will be combined and the definition of the join specification. The one exception is our *mutant* method (which allows the combination of several families of mutant programs) in which the combined clause needs to be approved by the user.

Our system requires only a small amount of program history information: the name of the program, the type of the program, the arity, the name of the skeleton, the number of clauses, a list containing the set of non mutant clauses and a list of mutant clauses. This *program history* is also recorded each time two programs are combined in case this combined program should be combined with another program. Also, this *program history* reduces the user interaction required to a minimum. The preconditions of the set of rules for deciding which method can be applied can be established automatically. An example of the program history is given in Section 5.

The selection procedure automatically selects a composition method according to the features of the two programs we want to combine, determined by the techniques used in their construction. For instance, two programs can be combined using the meta-composition method (explained in Section 4) if they have the following features:

- *the skeleton employed*: both programs should be constructed using the same skeleton — this skeleton is such that at least one of its clauses performs a data structure decomposition;
- *number of clauses*: both programs should have the same number of clauses as the original skeleton;
- *data structures*: programs should make use of the same data structure; and
- *pattern matching*: the pattern used to recurse up or down the data structure should be the same in both programs.

These features give rise to classes of programs. Our system, however, is not restricted to combining programs of the same class: programs of different classes can also be combined in a limited range of cases (see Section 4).

3.2 Composition System

The composition system allows the construction of more complex programs by combining simpler Prolog programs, previously built by means of the techniques editor. This composition system contains a set of methods for combining programs with the same flow of control or with a different flow of control. None of these methods represents a general solution but every method works efficiently for a specific class of programs (as determined by the selection system). Each method implemented in the composition system can be seen as a sequence of transformation operations. Thus the composition problem can be seen as a novel application of program transformation as opposed to its orthodox use in program synthesis. These methods are given to the user in an interactive system to help in the construction of a program that is appropriate for his/her application. The sequence of transformation operations applied to the initial programs is obtained from the transformation rules library. The transformed program has the same meaning but, in general, it will have better computational behaviour than the initial programs.

3.3 Transformation Rules Library

The proposed library of transformation rules holds a collection of transformation operations which preserve correctness. The set of rules defined in our system is given below.

A *join specification* is a formal expression in which we define the characteristics of the new program than can be generated using two other programs. In the join specification we are defining the number of parameters in the new procedure, data input vectors from each program and data output vectors. The join specification is defined as follows:

Let T be defined as $\mathcal{T} :- \mathcal{P}, \mathcal{Q}$ where $\mathcal{P}$ and $\mathcal{Q}$ are calls to predicates that will be joined; they are called join operands and $\mathcal{T}$ is the join target (the resulting joined procedure) [LS87]. An example of a join specification is in Section 5.

1. The *unfolding* operation can be defined as follows: let P be a program and C a clause in P of the form $A \leftarrow Q_1, B, Q_2$ where A and B are atomic formulae and Q_1 and Q_2 are conjunctions of literals. Let $H_1 \leftarrow R_1, \ldots, H_n \leftarrow R_n$ be those clauses in P whose heads H_i unify with B with the mgu's $\theta_1, \ldots, \theta_n$ (the most general unifiers). Unfolding B on C we obtain the clauses $(A \leftarrow Q_1, R_1, Q_2)\theta_1, \ldots, (A \leftarrow Q_1 R_n, Q_2)\theta_n$. By replacing C by these clauses we transform the program P into the program P' [FF91].

2. The *folding* operation is defined as follows: let P be a program and D a set of join specifications. Let C be a clause in P of the form $A \leftarrow Q\theta, R$ where Q and R are conjunctions of literals and θ is a substitution. Let C_1 be a clause of the form $H \leftarrow Q$ in D which is not a instance of C. Folding C using C_1 generates the clause C_2 of the form $A \leftarrow H\theta, R$ provided that unfolding C_2 on $H\theta$ with respect to D gives C; C_1 is the only clause in D whose head unifies with $Q\theta$ and θ maps variables which appear in Q, but not in H into distinct variables which do not occur in C_2. Replacing C by C_2 transforms P into P' [FF91].

3. The *unfold_B* operation is an extension of the previous unfold operation. This operation works for specifications defined as: $T :- P, Q, F_1 \ldots F_n$. where P and Q are join operands, T is the join target and $F_1, \ldots, F_n$ are subgoals which use the output results from predicate P and Q for performing new computations.

4. The *meta-folding* operation is similar to the folding operation but it can cope with local variables in the instances of the operands of the join specification. The *meta-folding* operation verifies, before folding, that the subgoals (from program P_1 and from program P_2), which are candidates for joining are enhancements of the same subgoal in the skeleton by using the *program history.* If both are enhancements of the same subgoal in the skeleton then the two local variables which control the recursion in program P_1 and program P_2 are unified, the subgoals are folded, and the binding of two variables is propagated through all points at which these local variables appear in the unfolded clause.

5. The operation *goal merge* produces a new program P' in which all the identical subgoals are replaced by a single occurrence [TS84].

6. Arithmetic rules are considered as useful transformations in the process of simplification of expressions in the domain of real numbers and natural numbers. These rules are, for instance, the identity element for addition, identity element for multiply, commutativity and associativity in real or natural numbers.

4 Methods

Our composition system is formed by a set of methods for combining Prolog programs with the same and different flows of control. The importance of classifying programs by restricting the sequence of transformation operations in each method is of particular importance because we can combine programs in more efficient forms by applying the most efficient method for each class. For instance, the combined program obtained using the *particular method* has better computational behaviour than the program produced using the *general method*, but it is only possible to apply the particular method to specific types of programs (see below).

Our set of methods is divided in two groups: one set is used for combining programs with the same flow of control and the second for programs with different flow of control for restricted classes of programs. The first group of methods (for combining programs with the same flow of control) is defined as follows:

1. the *synchronization* method handles the problem of joining programs which can be schematised in the following join specification:

   $$T(IP, IQ, OP, OQ) \text{ :- } P(IP, OP), Q(OP, OQ).$$

   where the input vectors of variables are IP and IQ and, similarly; OP and OQ are the output vectors of variables.

   The input for predicate Q is the output of predicate P. The method for combining this kind of program is to create a new program which is basically the join specification. This is the most primitive way of combining programs.

2. *join 1-1* combines programs with the same number of clauses. This method combines all the pairs of corresponding clauses of two programs [LS87]. This method works only with programs which are extensions of the same skeleton. This allows us to combine the ith-clause from program P_1 with the ith-clause from program P_2 where P_1 and P_2 are extension of skeleton S. Note that it is the fact that they are derived from the same skeleton (without changing its control flow) which makes this possible. This restriction avoids the generation of redundant clauses because clausal join (which combines pairs of clauses) operates on the same instance of the data structure (same form in the induction parameter).

3. *procedural-join* is an extension to the previous method. This method composes a new extension from two given programs by firstly taking clause number one of program P_1 with all clauses in program P_2 (taken one by one); secondly, taking clause two of program P_1 with all clauses in program P_2 and so on. This method allows the

combination of clauses which are not corresponding (from the skeleton to the program) but can only combine programs with the same number of clauses and the same form of structural induction parameter [LS87].

4. the *meta-composition* method combines programs by using a join specification and knowledge about the history of development of the program. In this method we generalise Sterling's algorithm [LS87] by relaxing the constraint that the two recursive calls folded away in the last step must have syntactically identical input variables. This is not always the case; for example, when the value being recursed over is itself computed by a call to a user-defined predicate. This method can be used for combining two classes of programs such as meta-interpreters and programs constructed using the *counter* skeleton. Also this method can be used for any class of program which is constructed by using data abstraction. The abstraction process replaces a sequence of subgoals that occur one or more times in a body of the clauses of a given program with a new predicate, and adds a clause whose head is the new predicate and whose body is the sequence of subgoals.

5. The *particular* method combines programs that satisfy the following restrictions in the join specification:

$$T(IP, IQ, OT) \ :\!-\ P(IP, OP), Q(IQ, OQ), F(OP, OQ, OT).$$

   such that both programs P and Q were constructed by applying the technique *count* or *sum* (at some stage of program development). F must compute values which involve the same arithmetic operands that are used to compute results in predicates P and Q. This method reduces the number of local variables by applying arithmetic rules and performing the folding operation using a join specification formed by more than two operands.

6. The *general* method uses the following join specification:

$$T(IP, IQ, OT) \ :\!-\ P(IP, OP), Q(IQ, OQ), F(OP, OQ, OT).$$

   where the restriction on F in the *particular* method does not apply. This method works by combining program P and program Q to produce a combined program P_Q and finally builds the combined program T which is as follows:

$$T(IP, IQ, OT) \ :\!-\ P\_Q(IP, OP, IQ, OQ), F(OP, OQ, OT).$$

The set of methods for programs with different flow of control are shown as follows:

1. The *mutant* method is used in the combination of programs with slightly different control flow. An example of the use of this method appears in Section 5. In this example `count/2` is created using the traverse list skeleton formed by two clauses and program `get_odd/2` is a mutant of the same skeleton (traverse list with two clauses).

2. The *traverse-short_traverse* method combines programs created by using the *traverse* and *short_traverse* skeletons. These programs are combined using procedural join because both skeletons have the following restrictions: the only difference between the skeletons *traverse* and *short_traverse* is the base case; the base case for the traverse skeleton ensures that the entire list will always be processed and in short_traverse the execution will either traverse the entire list or stop when a condition has been met.

3. The *traverse-search* method combines programs created using the *traverse* and *search* skeleton. These kinds of programs cannot combined in a single program but they can be combined by using our synchronization method in a new program formed by the calls to each program to be combined.

4. The *short_traverse-search* is the same case that the *traverse-search* above.

For the sake of brevity we only present two of our methods, called the *meta-composition* and *mutant* method, in the current paper. A full set is given in [VV94].

The algorithm for *meta-composition* uses as input the *join specification* and a pair of programs $(T, \langle P, Q\rangle)$, where T is the join specification and P and Q are predicates. The output from this algorithm is the combined program. The notation which will be used in the following algorithm is this: P_i is the ith-clause in program P, $P_{i,head}$ is the head of clause P_i and $P_{i,body}$ is the body of the ith-clause in program P. An example of how this algorithm works is presented in the next section.

1. Create an instance T_i of the join specification (T :− P, Q which the join specification provided by the user).

2. Unfold P and Q in T_i with respect to P_i and Q_i.

   - if P and $P_{i,head}$ unify with mgu θ_P and
   - Q and $Q_{i,head}$ unify with mgu θ_Q

   then replace P with $P_{i,Body}$ and Q with $Q_{i,body}$ in T_i, using the mgu $\theta_P\theta_Q$, to produce clause T_i:

   $$T_i \text{ :- } P_{i,body}, Q_{i,body}\theta_P\theta_Q$$

   If $P_{i,body}$ is a conjunction of n goals $P_{i,goal1}, ..., P_{i,goaln}$ with $n > 0$ and $Q_{i,body}$ is a conjunction of m goals $Q_{i,goal1}, ..., Q_{i,goalm}$ with $m > 0$ then the result of the unfolding is the clause:

   $$T_i \text{ :- } (P_{i,goal1}, ..., P_{i,goaln}), (Q_{i,goal1}, ..., Q_{i,goalm})\theta_P\theta_Q$$

3. Apply the meta-fold operation if there exists $P_{i,goalp}$ and $Q_{i,goalr}$ in the body of the clause T_i such that in the program history these subgoals are enhancements of the same subgoal in the skeleton. Then bind the local variables used for recursing up or down the same data structure (used in the subgoals $P_{i,goalp}$ and $Q_{i,goalr}$), and replace $P_{i,goalp}$ and $Q_{i,goalr}$ with T_i.

4. Repeat this process for each clause in program P and program Q.

The *mutant* method copes with the problem of extra clauses which do not have a corresponding parent clause by performing partial unfolding in just one of the operands of the join specification. This algorithm is also an extension of join 1-1 and procedural join in which the application of the unfolding operation is blindly applied, thus creating redundant clauses.

The algorithm creates an instance of the join specification T_k; a partial unfold operation (one operand of the join specification) is performed in predicate P or predicate Q and the values of variables are passed through the instance T_k. The selection of which predicate is unfolded is determined as follows: if the mutant clause belongs to program P then P is unfolded, otherwise predicate Q is unfolded. This process is repeated while there are mutant clauses in each program. The operation of unfolding is applied partially through the join specification, and the final mutant combined clause can be accepted or rejected by the user. User interaction is required at this stage in order to determine the behaviour of the combined clause. The decision about what functionality the combined clause should have cannot be automated by the system. It is not deterministic, since more than one option can be chosen. There are two ways to change the behaviour of the offered combined clause: one is that the variables passed from the first operand to the second operand (which is not unfolded) can be redefined according with the wishes of the user and, secondly, subgoals in the body of the clause can be changed if the user wants to conditionally terminate a clause. For instance, the user could redefine the value of the variable used to control the depth of the proof search space with the purpose of reducing the number of subgoals to be proved.

The mutant algorithm is as follows:

Assuming that P contains the mutant clauses:

1. Create an instance T_i of the join specification (T :– P, Q).
2. Unfold P in T_i with respect to P_i.
    - if P and $P_{i,head}$ unify with the substitution θ_P and P_i is defined as follows:

    $$P_i \ :\!-\ A_1, \ldots, A_n$$

    Then replace P with $P_{i,Body}$ with the substitution θ_P to produce T_i:

    $$T_i \ :\!-\ ((A_1, \ldots, An), Q)\theta_P \quad with \ \ n > 0$$
3. Apply the fold operation using the set of join specifications.
4. Ask the user to approve the proposed combined clause.
5. Repeat the same process for each mutant clause that appears in program P.

This process can be described in the same way for Q in the case that Q is the mutant clause. Note that, we have two cases: depending on which is the mutant clause, if the mutant clause belongs to program P, the values of the variables in the head of the clause T_i are passed to Q. In the second case, if the mutant clause belongs to program Q then the values of the variables in the head of the clause T_i are passed to P.

5 Example

This section shows two differents approaches for the combination of a pair of Prolog programs. Solution 1 follows the algorithm by Proietti et al. [PP92] which, although it produces an efficient combined program, requires user interaction a key stage in transformation and in Solution 2 we produce the same result fully automatically.

The Proietti et al. algorithm applied in Solution 1 is based in unfolding, folding and addition of new join specifications. The main characteristic of the algorithm is that it eliminates unnecessary variables. This algorithm requires three actions which need to be defined for the user: the introduction of a set of join specifications for the folding step, selection of the calls in the body of the clauses for unfolding stages and choice of the replacement laws to be applied. However the algorithm used in Solution 2 uses program history in order to reduce the user interaction.

Solution 1

Our initial programs are the programs `get_odd/2` and `count/2`. The program `get_odd/2` is a mutant program based on the traverse skeleton which obtains the odd numbers from a list and the program `count/2` is an extension of the traverse skeleton which counts the length of the list.

```
1:  get_odd([], []).
2:  get_odd([H|T], [H|O]) :-
        odd(H),
        get_odd(T, O).
3:  get_odd([H|T], O) :-
         even(H),
         get_odd(T, O).
```

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

A new program `count_odd/2` which obtains the odd numbers and computes the length of the list at the same time is generated by using the join specification defined as follows:

```
6:  count_odd(L, C) :- get_odd(L, Os), count(Os, C).
```

Firstly apply the unfold operation to the definition of `get_odd/2` in `count_odd/2`:

```
7:  count_odd([], C) :-
             count([], C).
8:  count_odd([H|T], C) :-
             odd(H),
             get_odd(T, Os),
             count([H|Os], C).
9:  count_odd([H|T], C) :-
             even(H),
             get_odd(T, Os),
             count(Os, C).
```

After unfolding the call `count/2` in clause 7 we obtain the following clause:

```
10:  count_odd([], 0).
```

By folding clause 9 we get the clause shown below. At this stage no new join specification is required:

```
11:  count_odd([H|T],C) :-
             even(H),
             count_odd(T,C).
```

A new join specification `new1/3` can be introduced in order to continue transforming the program. The new join specification is shown as follows:

```
D: new1(H,T,C) :- odd(H), get_odd(T, Os), count([H|Os], C).
```

Clause 8 is folded by using the new join specification `new1/3` obtaining the following clause:

```
8f:  count_odd([H|T], C) :- new1(H,T,C).
```

Unfolding the call `get_odd/2` in clause number `D`, we obtain the following clauses:

D_1: `new1(H,T,C) :- odd(H), get_odd(T, Os), count(Os, C1), C is C1+1.`

The fold operation is applied in clause D_1, obtaining the following clause:

D_2: `new1(H,T,C) :- odd(H), count_odd(T,C1), C is C1+1.`

At this stage the resulting program consists of clauses 10, 11, 8f and D_2.

A final simplification step is done by unfolding transient predicates, like new1/3. Unfolding new1/3 in clause 8f we obtain the following clause:

D_3: `count_odd([H|T],C) :- odd(H), count_odd(T,C1), C is C1+1.`

Finally the resulting combined program is formed by clauses 10, 11 and D_3.

Solution 2

The main difference between Solution 1 and 2 is that Solution 2 does not require the addition of new join specifications, using instead available information from the program history. In the third clause, only partial unfolding is performed (the second operand of the join specification count/2 is not unfolded). This relies on the fact that this is a mutant clause. The knowledge that the third clause is the mutant clause is automatically deduced by analysing the list of mutant clauses which are kept for the program `get_odd/2` in the program history.

The program history is defined using the predicate *his_prog/8*.

$$his\_prog(P, T, A, S, N, TsT, NM, M).$$

where P is the name of the program, T is the type of the program (one of the types in our classification), A is the arity, S is the name of the skeleton, N is number of clauses for the program, TsT is number of tests which hold the recursive cases, NM is a list containing the set of non mutant clauses and M is a list containing the set of mutant clauses. An instance of the program history is shown below for one of the programs `get_odd/2`, in our working example.

```
his_prog(get_odd,type_mutant,2,traverse,3,2,
            [(1,traverse(T),get_odd(T,O)),(2,traverse(T),get_odd(T,O))],
            [(3,traverse(T),get_odd(T,O)]).
```

Solution 2 is described as follows:

Firstly by applying the unfold operation in the definition of `get_odd/2` in `count_odd/2` we arrive at the same result as Solution 1 (clauses 7, 8 and 9).

Secondly we apply the unfolding operation to the calls `count/2` in the first and second clause. Those clauses that can be unfolded are determined by means of the program history. The parameter 8 in the relation his_prog/8 states that clause 3 is the mutant clause. Therefore this clause does not have corresponding clause from the second program (`count/2`), so the system decides to unfold the first operand P of the join specification $(T\ :-\ P, Q)$ and to spread the values through the second operand.

```
count_odd([], 0).
count_odd([H|T], C) :-
            odd(H),
            get_odd(T, Os),
            count(Os, COs),
            C is COs+1.
count_odd([H|T], C) :-
            even(H),
            get_odd(T, Os),
            count(Os, C).
```

Finally, applying the fold operation obtains the following clause:

```
count_odd([], 0).
count_odd([H|T], C) :-
            odd(H),
            count_odd(T, COs),
            C is COs+1.
count_odd([H|T], C) :-
            even(H),
            count_odd(T, C).
```

This approach is only possible if technique descriptions of programs are conveniently available from the techniques editor.

6 Conclusions

This paper presents a methodology for combining programs constructed by using a specialised techniques editor which records the program history. This program history is used to reduce user interaction in program transformation. Our example shows how to combine two programs with a slightly different flow of control. The first solution relies on the introduction of new join specifications in order to apply the fold operation and the second solution relies on the program history. Our conclusion is that, using a small amount of knowledge about the program called the program history, and with little user interaction, we can produce efficient combined programs. The described system has been implemented using SICStus Prolog (version 2.1.1) running under Unix using a Sun workstation.

Programs developed using conventional editors could be incorporated in this environment by having their components (skeletons and techniques) identified and their history extracted. One plausible approach is to analyze the program and find out which class (of our classification) it belongs to. This is presently being investigated.

Acknowledgements

We would like to give acknowledgements to Alberto Pettorossi, Wamberto Vasconcelos, Jeremy Crowe and Edward Carter for their useful comments on this paper and also to the anonymous LoPSTr-93 referees for their constructive criticisms.

References

[Bow93] Andy Bowles. A Techniques Editor for Prolog novices. Internal note submitted for publication, DAI, 1993.

[FF91] Norbert E. Fuchs and Markus P. J. Fromherz. Schema-Based Transformations of Logic Programs. In *Logic Program Synthesis and Transformation, Workshops in Computing*. Springer Verlag, 1991.

[GH91] Timothy S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20:173–192, 1991.

[KLS89] Marc Kirschenbaum, Arun Lakhotia, and Leon S. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Case Western Reserve University, 1989.

[LS87] Arun Lakhotia and Leon Sterling. Composing Logic Programs with Clausal Join. Tr 87-25, Computer Engineering and Science Department, Case Western Reserve University, 1987.

[PP92] Maurizio Proietti and Alberto Pettorossi. Best-first Strategies for Incremental Transformations of Logic Programs. In *Second International Workshop on Logic Program Synthesis and Transformation*, 1992.

[Rob91] Dave Robertson. A Simple Prolog Techniques Editor for Novice Users. In G. A. Wiggins, C. Mellish, and T. Duncan, editors, *3rd UK Annual Conference on Logic Programming*, pages 190–205. Springer Verlag, April 1991.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Sweden, 1984.

[VV94] Maria Vargas-Vera. *Guidance during Program Composition in a Prolog Techniques Editor*. PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1994. In preparation.