Building Large-Scale Prolog Programs using a Techniques Editing System

M. Vargas-Vera [†] W. W. Vasconcelos [*]
D. S. Robertson
Dept. Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN — Scotland, Great Britain

Abstract

We describe an integrated environment which addresses three distinct aspects of Large-Scale Prolog Programs: the formalization of the programming practices one should use in order to build reliable and maintainable programs, the computer-aided use of these practices to develop programs, and the combination of these programs into more sophisticated and efficient programs. We also propose the use of program histories (*i.e.*, the information of how programs were developed using a techniques-based editor) to improve the process of program combination.

1 Introduction

Programmers can improve their performance (and the quality of their programs) by making use of methodologies to help them write, document, test and maintain their programs. Prolog's simple (albeit compact) syntax is an attractive feature, but programmers constantly face tedious tasks (*e.g.*, writing large programs with similar procedures, or combining two similar programs together to improve their efficiency).

Kirschenbaum et al. proposed a methodology for developing logic programs called *stepwise enhancement* [KLS89] with which programmers could be guided in constructing modular large-scale Prolog programs. This methodology makes use of simple "pre-canned" components which programmers may experiment with different combinations — programmers are saved from the gory details and can concentrate on the relationships between these components. The stepwise enhancement methodology encourages software reuse and standardization and tools embodying it are easily implemented [Rob91].

Based upon this methodology we have implemented an automatic system to help the construction of modular large-scale Prolog programs making use of a techniques editing system and program combination. This integrated environment has the following features:

1. the ability to devise generic control flows (skeletons) which give rise to classes of programs and ways to enhance these control flows (techniques) making them more useful (*Prolog Techniques Meta-Editor*);
2. the ability to use the programming practices devised in the Meta-Editor to build programs (*Prolog Techniques Editor — PTE*);

[*] On leave from State University of Ceará, Ceará, Brazil; sponsored by Brazilian National Research Council (CNPq), under grant no. 201340/91-7.
[†] Sponsored by National University of Mexico

3. the ability to combine the programs devised using the Prolog Techniques Editor into more complex ones (*Combination system*).

In this paper we address two components of this environment: the Meta-Editor and the combination system. The Meta-Editor manages a comprehensive (and hopefully small) repertoire of techniques to be offered to users of the PTE, achieving a compromise between generality and feasibility to manage. We aim to provide a comprehensive set of techniques, but it may be too ambitious to try to provide a complete set of *every* technique a programmer needs — some of these are far too particular to be offered as a standard practice. Also, for the construction of more sophisticated programs the programmer may require specialized techniques useful for his/her particular needs. The Meta-Editor helps in this, providing a (constrained) medium through which an expert programmer can devise and test arbitrary techniques, leaving some "slots" undefined. Other programmers can then conduct experiments using the PTE, using different techniques and filling in their slots to meet the specific needs of the program they are trying to develop.

The composition system allows the reuse of software and the development of optimized complex programs which can be repeatedly combined, producing more complex programs. We aim at more sophisticated programs which are difficult to combine using conventional program transformations. This is achieved by employing information supplied by the PTE (the *program history*) regarding the flow of control of each program to be combined. Although progress was made in the problem of combining arbitrary programs, it is still the case that not all pairs of programs can be combined: only pairs of programs sharing certain structural features may be combined in a single program. Therefore we supply a classification hierarchy which allows appropriate combinations to be identified according to these features.

This paper is organized as follows: in Section 2 we describe in more detail the adopted stepwise enhancement methodology. Section 3 describes the proposed environment for constructing large-scale Prolog programs, concentrating on the Meta-Editor and the Combination System: the components are explained by means of a working example in which two programs are developed and efficiently combined together. Finally, Section 4 draws some conclusions, points some limitations of the proposal and gives directions for future research.

2 Stepwise Enhancement using Skeletons and Techniques

The *stepwise enhancement* methodology uses a set of syntactic entities, called *skeletons*, which describe the flow of control of the most commonly found Prolog programs, and a set of syntactic transformations (selection mappings), called *techniques*, to be applied to the skeletons. The techniques convey the programming practices (*e.g.*, adding parameters, binding them properly, adding goals or clauses) carried out by experienced programmers upon very simple programs processing input data (*e.g.*, traversing or manipulating lists) in a straightforward way.

The *modus operandi* of this approach suggests the idea of developing a program by finding the suitable basic control flow needed to solve the problem. Then, programmers choose from amongst the skeletons the one which best suits their needs. After the skeleton has been selected, the programmer customizes it to obtain the final program. This customization process is achieved by applying techniques to skeletons yielding *enhancements* (*i.e.*, original skeletons enhanced with extra arguments, goals and/or clauses). The enhancements can be re-used as skeletons allowing one to repeat the process of application of techniques until the final program is obtained. If the enhancement is such that the control flow does not change then we have an *extension* of the skeleton. If appropriate sets of skeletons and techniques are provided, the task of writing a program (largely informal and error-prone) could be transferred to that of finding a skeleton and a sequence of techniques.

We regard programming techniques as relationships between one or more additions and a skeleton. *Additions* are specifications of syntactic alterations to be carried out upon skeletons, inserting variables, constants and subgoals in order to devise more useful programs. Collecting values and building data structures with them, or counting the number of iterations in the loop are two examples of interesting computations that may be performed once the flow of control is established by a skeleton.

3 An Environment for Constructing Large-Scale Prolog Programs

We propose an integrated environment which addresses three distinct aspects of programming: the formalization of knowledge about how programs should be built (*meta-editor*), the development of programs themselves (*techniques editor*), and the composition of programs into more efficient combined forms (*composition system*). The *meta-editor* incorporates the

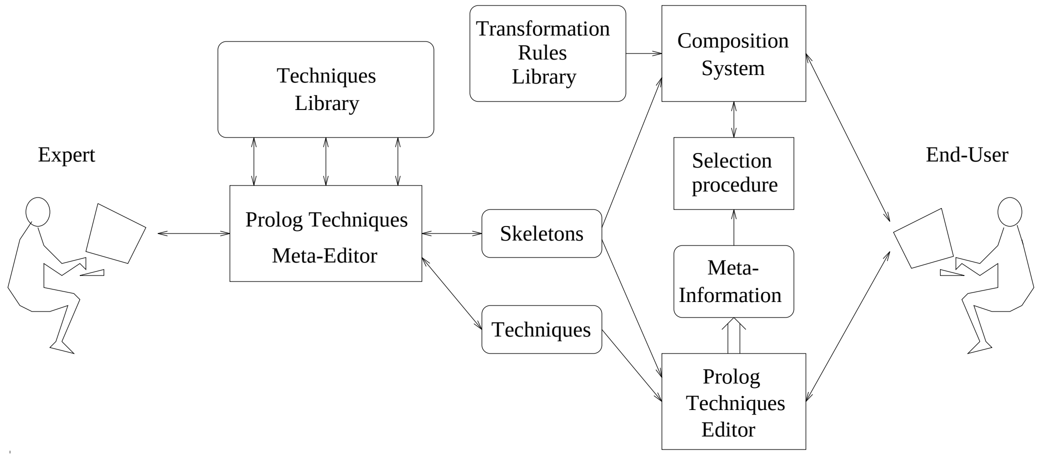

Figure 1: Architecture of Environment for Constructing Prolog Programs

formalization of programming techniques (which will be defined below) providing its users, expert programmers, with a set of operations by means of which skeletons, additions and techniques can be defined, tested and stored in a *techniques library*. The components of the techniques library are subsequently offered to the users of a *Prolog Techniques Editor* (PTE) for further customization and combination, according to the users' specific needs. The PTE is a knowledge-based software development tool providing its users with a set of skeletons and applicable techniques and guiding them through a series of interactions at the end of which a program (procedure) is obtained and tested. If the resulting behavior differs from the intended one, the user may backtrack to previous decision points and examine different alternatives [Rob91].

The *composition system* allows the construction of complex Prolog programs by combining simpler Prolog programs constructed using a techniques editor. The composition system uses the same set of skeletons provided by the meta-editor to the PTE, and a combination method obtained by *selection procedure* using the *meta-information* (*i.e.*, the *history*) of each program to be combined. The composition system yields the combined program and its history and stores them for future stages in the composition process.

3.1 The Meta-Editor: Designing Programming Techniques

Current attempts to describe Prolog programming practices remain largely informal. They have been described by means of examples (along with an informal explanation in some natural language) showing instances of their successful applications. We propose a way to represent these programming practices, and to make this representation available to expert programmers by means of an automated tool, the Prolog Techniques Meta-Editor. The motivation for this endeavor is that of enabling expert programmers to describe (although in a constrained medium) the programming practices they make use of while building programs and to make these practices available to other programmers (experts or not) by means of a PTE. We intentionally keep our notation close to standard Prolog (allowing syntactic operations such as renaming a predicate and removing arguments) avoiding the intricacies of higher-level languages.

Skeletons

Skeletons are basic syntactic constructs depicting the flow of control of a Prolog program. They are basic Prolog programs themselves processing input data in a simple way, and are used to build more complex (and more useful) programs by establishing a flow of control which will be followed while computation takes place. More formally, a skeleton `S` is a set $\{s_1, \ldots, s_n\}$ of clauses $s_i, 1 \leq i \leq n$, of the following form:

```
skel(X):-
   guard(I,X),
   obtain(I,X,Y0,...,YM,Z0,...,ZR),
   offer(Y0,...,YM),
   skel(Z0),...,skel(ZR).
```

At least one clause s_i must be such that `R` $\neq 0$; that is, there is at least one recursive clause. Variables `Y0,...,YM` are named *non-recursive* values and, if `M = 0`, they can be absent. Variables `Z0,...,ZR` are named *recursive* values and are found in the same number of recursive calls. Base-case clauses are a special case of the recursive clause, when `R` $= 0$.

Subgoal `guard(I,X)` unifies with the head goal of a clause of form

```
guard(I,X):- test(1,X),...,test(T,X)
```

where each `test(J,X)` is either:

- a *constant-dependent test* (*e.g.*, `X = C`, where `C` is a constant);
- a *system single-argument test* (*e.g.*, `atom(X)`);
- a *user-defined test*; or
- `true` (in which case it is removed altogether).

Subgoal `obtain(I,X,Y0,...,YM,Z0, ...,ZR)` unifies with the head goal of one of the following clauses:

- *Data structure decomposition* – where `F` is a functor:

```
obtain(I,X,Y0,...,YM,Z0,...,ZR):- X =..  [F,Y0,...,YM,Z0,...,ZR].
```

- *Calculation* – `FJ` and `GK` are expressions in terms of `X` obtained by means of `expressions`:

```
obtain(I,X,Y0,...,YM,Z0,...,ZR):-
                  expressions(X,F0,...,FM,G0,...,GR),
                  Y0 is F0,...,YM is FM,
                  Z0 is G0,...,ZR is GR.
```

- *User-defined Process* – `Pred` is any user-defined predicate:

```
obtain(I,X,Y0,...,YM,Z0,...,ZR):- call(Pred(X,Y0,...,YM,Z0,...,ZR)).
```

It may be that not all arguments are used in the predicate definition, in which case the spurious arguments are removed. The `offer` predicate is a repository of non-recursive values (to be employed by some addition — see below) which can be customized by the users of the PTE.

The Meta-Editor guides its users through a series of interactions in which a generic template embodying the definition above is gradually customized. As an example, skeleton `n_solve` below may be obtained by having the expert programmer define the number of non-recursive and recursive clauses, the number of variables of each clause and the process of obtaining new values (second clause). This skeleton describes the flow of control of programs belonging to the useful class of meta-interpreters.

```
n_solve(X):-
      guard(1,X),
      obtain(1,X,Y1),
      offer(Y1).
n_solve(X):-
      guard(2,X),
      obtain(2,X,Y1,Y2,Z1),
      offer(Y1,Y2),
      n_solve(Z1).
n_solve(X):-
      X = (Z1,Z2),
      n_solve(Z1), n_solve(Z2).
```

By means of the PTE, (less experienced) programmers can make use of the `n_solve` skeleton and adapt it to their needs: predicates `offer` can be changed into other predicate symbols, if these new symbols do not change the control flow of the skeleton (no tests or recursive calls are allowed) and adjacent subgoals `guard` and `obtain` can be merged together and later renamed. Variable and predicate symbols can also be changed. The clauses may have their relative positions changed, but clauses cannot be deleted or added. Procedure `interpret/1` below was devised using `n_solve`:

```
interpret((Proof1,Proof2)):-
      interpret(Proof1), interpret(Proof2).
interpret(Proof):-
      fact(Proof,Fact),
      nl, write([Fact,' is a fact in the database']).
interpret(Proof):-
      rule(Proof,Head,Body,Proof1),
      nl, write([Head,' is proved using rule']).
      display_rule(rule(Head,Body)),
      interpret(Proof1).
```

The last clause of `n_solve` was transposed as the first clause of `interpret`, the variable symbols were changed, `guard` and `obtain` predicates were merged together and then substituted and `offer` predicates were changed into user-defined predicates and system predicates.

Additions

Additions are represented as single-argument Prolog procedures. However, they have different patterns for their clauses depending on their usage mode (input or output). An *input mode addition* A^+ is a set of clauses $\{a_1, \ldots, a_n\}$ of the form

```
add(X):-
   request(Y0,...,YM),
   prepare(_,X1,X,Y0,...,YM), offer(X1),
   prepare(0,Z0,Y0,...,YM), offer(Z0), add(Z0),
      ⋮
   prepare(R,ZR,Y0,...,YM), offer(ZR), add(ZR).
```

An *output mode addition* A^- is a set of clauses $\{a_1, \ldots, a_n\}$ of the form

```
add(X):-
   request(Y0,...,YM),
   add(Z0),...,add(ZR),
   prepare(_,X,Y0,...,YM,Z0,...,ZR).
```

Each defined addition must contain at least one recursive clause the reason being that they will be applied to skeletons which have this restriction. Variables `Y0,...,YM` are named the *loose-ends* `E` of the clause, and if `M = 0`, they may not exist. The base-case clause is a special form of the recursive one, when `R = 0`. Predicate `prepare` unifies with the head goal of one of the following clauses:

- *Compose Data Structure* – where `F` is a functor:

  ```
  prepare(I,X,Y0,...,YM):- X =..  [F,Y0,...,YM,Z1,...,ZR].
  ```

- *Calculation* – `Exp` is an expression in terms of `Y0,...,YM` obtained by means of `express-ion`:

  ```
  prepare(I,X,Y0,...,YM):- expression(Y0,...,YM,Exp), X is Exp.
  ```

- *User-defined Process* – `Pred` is any user-defined predicate:

  ```
  prepare(I,X,Y0,...,YM):- call(Pred(X,Y0,...,YM)).
  ```

`A` will be used to denote an arbitrary addition (either input or output mode). Some of the arguments in the predicates may not be used. When $\mathtt{R} = 0$, the clause is responsible for the adjustments carried out on non-recursive clauses of skeletons or enhancements; when $\mathtt{R} \neq 0$ the clause describes the alterations to be performed on recursive clauses. Different recursive clauses in additions handle clauses in skeletons with different numbers of recursive calls. The predicate `request` is simply a repository of variables `Y0,...,YM`, the loose ends of the addition, and needs no definition. Once the loose ends are bound to constants or other variables in different additions or skeletons, they provide the integration between the components of a programming technique, as will be seen below.

For instance, the Meta-Editor supports the design of output mode addition `add_one (-)` shown below, by having the expert programmer deciding upon the number of clauses (3), the number of requested values (a constant for the base-case clause and no requested values for the recursive clauses) and of recursive calls (one base-case clause, and two recursive clauses with one and two recursive calls, respectively), and the definition of each `prepare` predicate:

```
add_one(X):-
   X = 1.
add_one(X):-
   add_one(Z1),
   X is Z1 + 1.
add_one(X):-
   add_one(Z1), add_one(Z2),
   X is Z1 + Z2.
```

This addition is offered to the users of the PTE who can only rename variables and apply the addition to those skeletons and enhancements compatible with it. For instance, given the customized version `count/1` of `n_solve` shown on the left-hand side below, the result `count/2` of applying addition `add_one` to it is shown on the right-hand side:

```
count((A,B)):-                 count((A,B),C):-
   count(A),                      count(A,CA),
   count(B).                      count(B,CB)
                                  C is CA + CB.
count(A):-                     count(A,1):-
   fact(A,R).                     fact(A,R).
count(A):-                     count(A,C):-
   rule(A,Head,Body,B),           rule(A,Head,Body,B),
   count(B).                      count(B,CB),
                                  C is CB + 1.
```

Techniques

Explicit relationships drawn among skeletons and additions give rise to the programming techniques which will be offered to the users of the PTE. Two clauses are said *compatible* when they are both base-case clauses or when they have the same number of recursive calls. A skeleton S (or an addition A) is compatible with addition A' if for every clause in S (or A) there is exactly one compatible clause in A'. The *set of links* L_i between an addition clause $a_i \in \mathsf{A}$ with a set E_i of loose ends (those variables $X \in$ `request`) and a compatible clause $c_j \in \mathsf{C}$, with a set O of offered variables (those variables $Y \in$ `offer`) is a finite set of pairs E/O, where $E \in \mathsf{E}_i$ and $O \in \mathsf{O}$ or a constant. a_i/L_i denotes clause a_i with its loose ends $E \in \mathsf{E}_i$ replaced by $O \in \mathsf{O}$ (or by a constant) and removed from the `request` predicate. The symbol L (without subscripts) stands for the links $\mathsf{L}_1, \ldots, \mathsf{L}_n$ of an addition A.

The links are supplied by expert programmers and convey the knowledge used to integrate the independently devised pieces of a programming technique. Each variable in a loose end of an addition should appear exactly once in the set L of links to enable the use of the addition. The set of links can be provided in a generic way if loose ends are undetermined — they should then be linked to an undetermined (generic) number of offered variables. If the predicate `request` eventually runs out of loose-end variables, then it should be removed altogether. The Meta-Editor offers commands by means of which the loose ends of additions can be linked together.

The *composed addition* $\hat{\mathsf{A}}$ between compatible additions A' and A with a set of links L is $\hat{\mathsf{A}} = \mathsf{A}' \cdot \mathsf{A}/\mathsf{L}$. The composition of additions can be repeated any number of times, allowing previously composed additions to be further combined. As an example, additions `add_one` `(+)` and `carry (-)` shown on the left-hand side below can be combined together, given the set of links $\mathsf{L} = \{\{\texttt{B2/A1}\}, \emptyset\}$, into composed addition `add_carry` shown on the right-hand side below:

```
add_one(A1):-              carry(B1):-         add_carry(D1,D2):-
   offer(A1).                 request(B2),        offer(D1),
                              B1 = B2.            D2 = D1.
add_one(A1):-              carry(B1):-         add_carry(D1,D2):-
   A2 is A1 + 1,              carry(B1).          D3 is D1 + 1,
   offer(A2), add_one(A2).                        offer(D3), add_carry(D3,D2).
```

This composed addition uses an accumulator pair to count the iterations of the loop and to return the final value when the loop finishes. This example is provided to illustrate the definition of composed additions, however it shall not be used in our working example.

A *programming technique* T is the triple $\mathsf{T} = \langle \mathsf{S}, \hat{\mathsf{A}}, \mathsf{L} \rangle$ where S is a skeleton, $\hat{\mathsf{A}}$ is a composed addition compatible with S and L is a (possibly empty) set of links between the clauses of S and $\hat{\mathsf{A}}$. If L is empty, then the technique is said to be *skeleton-independent*. For example, the composed addition *add_carry* above is a (skeleton-independent) technique applicable to any terminating single-recursive skeleton. A non-composed addition can be a technique as well: `add_one (-)` shown above is one example.

3.2 The Composition Module: Combining Programs

The problem of combining two arbitrary Prolog programs in a single (more) efficient program may become too complex to be fully automated. We need to guarantee that, given two programs P_1 and P_2, the combined program P_3 has the same meaning, *i.e.*, the set of ground atoms derivable from the program, as the combined meaning (union) of the initial programs.

Programs, however, can be classified according to distinctive features and specific methods for combining them can be provided. For some of these classes, the combination of their programs can be greatly improved by making use of extra information — if the initial programs are constructed by means of techniques-based editors, then their *history* of development can be employed to obtain a more efficient combination. Also, this information can help in deciding which combination method should be used.

3.3 Selection Procedure

We have defined classes of programs and for each of them we propose a method of combination. The selection procedure automatically selects a composition method according to the features of the programs, determined by the techniques used in their construction. Our selection procedure uses as input information concerning the two programs the user wants to combine, and yields the method which will be used for combining these programs. For instance, two programs can be combined using the Meta-Composition method (explained below) if they have the following features:

- *the skeleton employed*: both programs are 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;
- *the tests*: at least one of the tests of corresponding clauses should be trivial (*i.e.*, logically equivalent to true) or both tests should be identical;
- *data structures*: programs should make use of the same data structure; and
- *pattern matching*: the pattern used to compose or decompose the data structure should be the same in every program.

Examples of programs complying with these constraints are `interpret/1` and `count/2`. These features give rise to classes of programs — our system, however is not restricted to combining programs of a same class: programs of different classes (sometimes) can also be combined. So far, ten classes have been catalogued [VVRI93].

3.4 The 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 identical or different (but compatible) skeletons. None of these methods represents a general solution but each method works efficiently for a specific class of programs.

Each method available 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 offered to end-users in an interactive system helping the combination of programs — some of these methods are fully automatic, while others are semi-automatic.

Each method in our composition system works by applying a sequence of transformation operations to the initial programs. These transformation operations are obtained from the transformation rules library. Each transformation operation is sound and ensures that the transformed program has the same meaning as the combination of the meanings of the initial programs, but in general it will have better computational behavior that the sequential execution of initial programs. For the sake of brevity only one method is presented: the Meta-Composition one. This method will be put to use in our working example.

3.5 The Transformation Rules Library

The proposed library of transformation rules holds a collection of transformation operations which preserve correctness. In this library we have several kinds of rules (unfolding, folding, meta-folding, goal merge plus rules of an arithmetic domain such as commutativity, associativity, etc.). Some of these operations (used in the meta-composition method) are:

1. The *unfolding* operation replaces a subgoal A in the body of a clause with the body of another clause the head of which is unifiable with A [TS84];

2. The *folding* operation replaces subgoals $G_1, \ldots, G_n$ (instances of the join operands) in the body of a clause with the head of another clause the body of which is unifiable with subgoals $G_1, \ldots, G_n$ [TS84];

3. The *meta-folding* operation is similar to *folding* but deals with local variables in instances of the operands of the join specification. The *meta-folding* operation verifies, before folding, that the corresponding subgoals of the programs to be combined are enhancements of the same subgoal in the skeleton by using the *program history.* If both subgoals are enhancements of the same subgoal in the skeleton then the variables controlling the recursion of the combining programs are bound, the subgoals are folded, and the binding of these variables is spread throughout the unfolded clause. If the corresponding subgoals are not enhancements of the same skeleton subgoal, then any subgoal is folded. An example of how this operation works is given in Subsection 3.6 below.

   The meta-folding operation binds existential variables — those appearing only in the body of clauses in composition or decomposition of data structures. One restriction on the application of the meta-folding operation is that both programs should use the same pattern to compose or decompose data structures.

A *join specification* [LS87] is an expression defining the characteristics of the new program to be obtained combining two other programs.

The Meta-Composition Method

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 programs. The output for this algorithm is the combined program. The notation which will be used in the following algorithm is: 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 subsection.

1. Create an instance T_i of the join specification $(T \text{ :- } P, Q)$.

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

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

   Then replace P with $P_{i,Body}$ and Q with $Q_{i,body}$ in T_i by using the mgu $\theta_P \theta_Q$ to the produced 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 exist $P_{i,goalp}$ and $Q_{i,goalr}$ in the body of the clause T_i such that in the program history we have the case that these subgoals are enhancements of the same subgoal in the skeleton. Then bind the recursive variables of the two 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.

3.6 Combining Programs using the Meta-Composition Method

Our initial programs are the meta-interpreters `interpret/1` (taken from [SS86]) and `count/2` which are constructed by using the skeleton `n_solve/1` (depicted above and also in [SS86]). The program `interpret/1` interprets the proof tree created for a goal (query) and `count/2` counts the number of rules needed for proving this goal — the definitions for the remaining predicates can be found in [SS86]. For the sake of explanation we shall consider `interpret/1` as being comprised of clauses P_1, P_2 and P_3 and `count/2` by clauses Q_1, Q_2 and Q_3. Additionally, we need the following definitions for `fact/2` and `rule/4`:

```
fact((Fact :- true),Fact).

rule((Goal :- Proof),Goal,Body,Proof) :-
     Proof \== true, extract_body(Proof,Body).
```

A new program `int_count/2` (comprised by clauses T_1, T_2 and T_3) which interprets a proof tree and simultaneously counts the number of rules in the proof is generated by using the following join specification:

```
int_count(Proof,C) :-  interpret(Proof), count(Proof,C).
```

The process consists of taking clause 1 from program `interpret/1` and clause 1 from program `count/2`, and creating an instance of the join specification (*i.e.*, $T_1 \;:-\; P_1, Q_1$). After unfolding `interpret(Proof)` and `count(Proof,C)` with respect to P_1 and Q_1, P_1 unifies with `interpret(Proof)` by means of mgu

$$\theta_P = \{(\texttt{Proof1}, \texttt{Proof2})/\texttt{Proof}\}$$

and Q_1 unifies with `count(Proof,C)` by means of mgu

$$\theta_Q = \{(\texttt{Proof1}, \texttt{Proof2})/\texttt{Proof}, \texttt{C}/\texttt{C}\}$$

Then by replacing $P_{1,body}$ and $Q_{1,body}$ in T_1, we obtain the following clause:

```
T1 :  int_count((Proof1,Proof2),C) :-
            interpret(Proof1), interpret(Proof2),
            count(Proof1,CP1), count(Proof2,CP2),
            C is CP1+CP2.
```

Applying the folding operation we get the following clause:

```
T1 :  int_count((Proof1,Proof2),C) :-
            int_count(Proof1,CP1), int_count(Proof2,CP2),
            C is CP1+CP2.
```

Taking clause 2 from each program and applying the unfolding operation we obtain the clause shown below:

```
T2 :  int_count(Proof,1):-
            fact(Proof,Fact1),
            nl, write([Fact1,' is a fact in the data base']),
            fact(Proof,Fact2).
```

In this clause we cannot simply fold the subgoals `rule(Proof,Head1,Body1,Proof1)` and `count(Proof2,CB)` in a single subgoal `int_count(Prof1,CB)` in the normal way by using the join specification, because `Proof1` is not the same variable as `Proof2`. The histories of the programs provided by the PTE inform us that subgoals `fact(Proof,Fact1)` and `fact(Proof,Fact2)` are enhancements of the same subgoal in skeleton `n_solve` (the merging of subgoals `guard` and `obtain`). This information enables us to infer, without having to perform any program analysis, which variables have the same functionality and hence can be bound together. In this case, we are able to determine that variable `Fact1` is the same as `Fact2` and we get a more optimized clause T_2:

```
T2 :  int_count(Proof,1):-
            fact(Proof,Fact1),
            nl, write([Fact1,' is a fact in the data base']).
```

Taking clause 3 from each program and Finally, taking clause 3 from each program and applying the unfolding operation we get to clause T_3:

```
T3 :  int_count(Proof,C) :-
            rule(Proof,Head1,Body1,Proof1),
            nl, write([Head1,' is proved using the rule']),
            display_rule(rule(Head1,Body1)),
            interpret(Proof1),
            rule(Proof,Head2,Body2,Proof2),
            count(Proof2,CB),
            C is CB+1.
```

This clause can be optimized by means of the meta-folding operation. This operation determines that the variables Proof1 and Proof2 can be unified through the program history: these variables have the same functionality, *i.e.*, they are used to decompose the data structure in each program. The binding {Proof1/Proof2} of existential variables allows the folding of the subgoals rule(Proof,Head1,Body1,Proof1) and count(Proof2,CB) and the removal of the second rule/4 subgoal. The optimized combined clause T_3 is as follows:

```
T3 :  int_count(Proof,C):-
            rule(Proof,Head1,Body1,Proof1),
            nl, write([Head1,' is proved using the rule']),
            display_rule(rule(Head1,Body1)),
            int_count(Proof1,CB),
            C is CB+1.
```

The resulting combined program using the extra knowledge is shown below. This program has better computational behavior (traverses the proof a single time) than the combined program generated without knowledge:

```
int_count((Proof1,Proof2),C) :-
            int_count(Proof1,CA), int_count(Proof2,CB),
            C is CA+CB.
int_count(Proof,1):-
            fact(Proof,Fact1),
            nl, write([Fact1,' is a fact in the data base']).
int_count(Proof,C):-
            rule(Proof,Head1,Body1,Proof1),
            nl, write([Head1,' is proved using the rule']),
            display_rule(rule(Head1,Body1)),
            int_count(Proof1,CB),
            C is CB+1.
```

It could be argued that another way to find out that Head1, Body1 and Proof1 can be unified to Head2, Body2 and Proof2 respectively, is by doing an analysis of each program to be combined and performing mode and type analysis in rule/4 and display_rule/1. In more sophisticated meta-interpreters this analysis may turn out to be rather complex and expensive, whereas in our approach, all the required information is already available.

4 Conclusions and Directions of Research

This paper presents an integrated approach for building and combining Prolog programs. Programs are built by means of a Prolog Techniques Editor, using skeletons and techniques prepared by means of the Prolog Techniques Meta-Editor. The combination of programs uses the meta-folding operation, an extension to traditional transformation operations. Our claim is that by using knowledge about the programs to be combined (the *history* of each program) we can produce better composed programs. Sometimes, it is also possible to combine programs which cannot be combined without the use of this knowledge. The described system has been implemented using SICStus Prolog (version 2.1.1) running under Unix using a Sun workstation.

Cuts were left out of our system: no skeletons or techniques can be devised making use of it and the composition system also falls short in combining programs using cuts. Future extensions should offer means to devise skeletons and techniques with sensible cuts and enable the combination of programs build using them.

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.

References

[KLS89] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Computer Engineering and Science Department, Case Western Reserve University, Ohio, U.S.A., 1989.

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

[Rob91] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland, April 1991. Springer-Verlag.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[TS84] H. Tamaki and T. Sato. Unfold/fold Transformations of Logic Programs. In *2nd International Conference on Logic Programming*, 1984.

[VVRI93] M. Vargas-Vera, D. Robertson, and R. Inder. An Environment for Combining Prolog Programs. Research Paper 610, Department of Artificial Intelligence, University of Edinburgh, 1993.