Enhanced Schema-Based Transformations for Logic Programs and their Opportunistic Usage in Program Analysis and Optimisation*

*Wamberto W Vasconcelos***, *Norbert E Fuchs*
Institut für Informatik
Universität Zürich
Switzerland
`{vascon,fuchs}@ifi.unizh.ch`

A program schema is a generic description of a program in a suitable language where its important features can be highlighted and irrelevant details disregarded. In this research note we introduce some enhancements made to the proposal of using program schemata to guide program transformations. These transformations are syntactic alterations carried out in logic programs preserving the meaning of the initial program and improving its execution. By allowing the reference to any argument position within a predicate and introducing extra facilities to address common constructs in a generic fashion a greater number of programs can be abstracted by the same schema. We also describe in this note a realistic context to perform program transformations in which an opportunistic approach is taken during the analysis of a program: every possible way to improve it is attempted, either by examining each procedure in an isolated fashion, or by checking for conjunctions within clauses that can be used as joint specifications.

1 Introduction

The clean declarative semantics of logic programs hide many performance issues which must be accounted for if the program is to be executed using the currently available technology. These efficiency issues can be dealt with either at an early stage, during the preparation of the program, or at a later stage, when a complete, possibly inefficient, logic program is analysed and its inefficient constructs are detected and appropriately modified, yielding a program with a better computational performance.

If the first approach is taken and performance issues are to be considered during the preparation of the program then standard logic programming constructs, also named *programming techniques*, which guarantee a good computational behaviour to those logic programs incorporating them, can be used. Prolog programming techniques have been extensively studied (e.g. [Bowles et al. 94], [O'Keefe 90], [Ross 89] and [Sterling & Shapiro 94]); some programming techniques have been assigned names, such as *accumulator pairs* and *difference lists*, and are now part of the folklore of the logic programming community. It has been advocated that these standard practices should be explicitly taught as part of a discipline of methodical logic programming development [Kirschenbaum et al. 94; Sterling & Kirschenbaum 91] and logic programming environments have been implemented (e.g. [Bowles et al. 94] and [Robertson 91]) incorporating them.

The second approach addresses efficiency issues after the program has been devised, trying to optimise the existing code. This involves the analysis and subsequent transformation of a given, possibly inefficient, logic program into an equivalent version with better computational behaviour. The analysis and transformation tasks should be as automated as possible, thus relieving logic programmers from the extra burden of worrying about performance issues. An underlying

---

* This work was carried out within the European Human Capital and Mobility Scheme (contract No. CHRX-CT 93-00414/BBW 93.0268) while the first author was visiting the Institut für Informatik, Universität Zürich.

** PhD Student, Department of Artificial Intelligence, University of Edinburgh, Scotland, U.K., sponsored by the Brazilian National Research Council (CNPq), Grant No. 201340/91-7, while on leave of absence from UECE/Ceará, Brazil.

assumption of this approach is that commonly occurring but computationally inefficient constructs in logic programs can be identified and eliminated.

A third option combines both previous approaches: individual procedures[1] are devised using programming techniques and their interrelations are, whenever possible, improved by a program transformation system.

The work presented here fits into the second line of research above: we propose an approach to examining a given logic program and detecting opportunities to improve its computational efficiency. We have adopted the syntax of Edinburgh's Prolog [Clocksin & Mellish 87] and its execution model based on SLDNF-resolution [Lloyd 93]. Readers are, however, encouraged to envisage the generality and adaptability of the concepts presented here.

Our proposal consists of analysing a given Prolog program searching for opportunities to improve it. These opportunities are generic descriptions of commonly occurring inefficient programming constructs and how they should be changed in order to confer a better computational behaviour to the program. In order to describe these programming constructs we employ a *schema language*.

Changes to program constructs are depicted by means of *schema-based transformations*. A schema-based transformation consists of an inefficient Prolog programming construct and a sequence of prescribed modifications that would provide the construct with a better computational performance. A given Prolog program is scanned and its constructs are matched against the description of inefficient constructs of the schema-based transformations: if the matching occurs then the prescribed improvements are made and other opportunities in the program are further searched for. Each transformation must be formulated so as to describe Prolog constructs in an economic yet general fashion. We suggest an enhanced version of Gegg-Harrison's *program schemata* [Gegg-Harrison 89; Gegg-Harrison 91] for this purpose, following the proposal by Fuchs and Fromherz [Fuchs & Fromherz 91]. A program schema is an abstract description of a class of programs in which important aspects are highlighted and irrelevant details are disregarded.

In the next section we explain program schemata in more detail. In Section 3 we describe how program schemata have been used to guide program transformations; in Section 4 we show how they can be given more expressiveness by a simple notational enhancement and discuss the computational costs of such enhancements. In Section 5 we explain a realistic scenario in which the enhanced schema-based transformations can be used to analyse and improve the computational performance of a program. In Section 6 we describe our approach to deal with those situations when more than one program transformation can be used at one time. In Section 7 we draw some conclusions and show directions for future research.

2 Program Schemata

A program schema is a generic description of a program, or of a class of programs, in a suitable Horn-clause notation. Gegg-Harrison [Gegg-Harrison 89; Gegg-Harrison 91] proposed such a notation for Prolog programs and this has been adopted in our work, with some additional features explained below. The language of Gegg-Harrison's schemata allows the convenient expression of many Prolog programming constructs. Let there be the following schema

```
schema_A([],«&1»).
schema_A([Head|Tail],«&2»):-
      <pre_process(«&3»,Head,«&4»),>
      schema_A(Tail,«&5»)
      <,post_process(«&6»,Head,«&7»)>.
```

where the single angle brackets "<…>" represent optional constructs, the constructs "**«&*n*»**", called *schema variables*, stand for possibly empty sequence of arguments the details of which are not

[1]We shall employ Deville's [Deville 90] definition of a *logic procedure* or simply *procedure* as the non-empty sequence of program clauses with the same predicate p/n (predicate symbol p with number of arguments n) in the head of each of these clauses.

relevant to characterise the form of the program. The schema above represents those programs whose first argument is a list and all its elements are recursively manipulated.

Although **schema_A** above economically describes a number of well-known programs such as *append/3*, *length/2*, and so on, it poses the unnecessary restriction of having the list argument always in the first parameter position of the predicate. A straightforward enhancement can be achieved by inserting schema variables before the list argument being focused upon, that is

```
schema_A*(«&1»,[],«&2»).
schema_A*(«&3»,[Head|Tail],«&4»):-
      ‹pre_process(«&5»,Head,«&6»),›
      schema_A*(«&7»,Tail,«&8»)
      ‹,post_process(«&9»,Head,«&10»)›.
```

However, a problem is introduced here: Gegg-Harrison's notational conventions are such that no constraints are imposed on the schema variables and hence it is not clear in the schema above if the same argument position is being uniformly addressed throughout the schema. In order to make sure that the same argument position containing a list is being considered in both clauses and in both the head goal and recursive call of the second clause, an extra notational device must be used to ensure that the size of the sequences of arguments **«&1»**, **«&3»** and **«&7»** is the same. This is one of the enhancements to Gegg-Harrison's schema language proposed in this paper and explained below.

3 Schema-Based Transformations of Logic Programs

Proposals for logic program transformation, such as [Lakhotia & Sterling 90], [Nielson & Nielson 90], [Pereira & Shieber 87] and [Proietti & Pettorossi 90], based on unfold/fold rules [Tamaki & Sato 84], all require human intervention at some point during the transformation process. The extent to which the human participation affects the performance of the program transformation varies according to each proposal, but they all share the common feature of performing only localised changes without a broader picture of the problem. The nature of the human intervention in these approaches requires much insight on specific features of the program and characteristics of the fold/unfold transformations.

Fuchs and Fromherz [Fuchs & Fromherz 91] have proposed the notion of schema-based program transformations, in which Gegg-Harrison's program schemata are used to characterise an actual context whereby efficiency-improving alterations can be performed. The transformations depict commonly occurring but inefficient Prolog constructs in an abstract manner, and prescribe the way they should be changed in order to confer a better computational performance on the program. These are concrete situations, described in a generic but complete fashion, to perform logic program transformations.

Schema-based transformations are standard syntactic manipulations of portions of a given program. In these transformations the program schemata depict both input programs and their improved output versions. The changes prescribed by the transformation comprise a complete sequence of transformation steps, involving the definition of new predicates (eureka rules), the order and choice of literals to apply unfold/fold rules, change of subgoals order, and any other relevant changes to the program in question. Schema-based transformations can thus be seen as a form of representation of knowledge about logic programs and how they sometimes can be changed to achieve a better computational behaviour. More formally, a schema-based transformation T is a quadruple of the form

$$\langle \langle G_1,\ldots,G_n\rangle, \langle S_1,\ldots,S_n\rangle, \langle H_1,\ldots,H_m\rangle, \langle T_1,\ldots,T_m\rangle \rangle$$

where $\langle G_1,\ldots,G_n\rangle$ and $\langle H_1,\ldots,H_m\rangle$ are conjunctions of subgoals, and $\langle S_1,\ldots,S_n\rangle$ and $\langle T_1,\ldots,T_m\rangle$ are respectively input and output program schemata. Such formalisation is to be understood as "given the conjunction $G_1,\ldots,G_n$, such that their definitions are respectively $S_1,\ldots,S_n$, then $G_1,\ldots,G_n$ can be replaced by $H_1,\ldots,H_m$ such that their definitions are respectively $T_1,\ldots,T_m$". The conjunction

$G_1,\ldots,G_n$ provides the context in which the transformation can take place; $S_1,\ldots,S_n$ and $T_1,\ldots,T_m$ are sequences of program schemata, describing, in an abstract fashion, the form the input and output programs should have.

Given a schema-based transformation T of the form above, the context of its successful application is a Prolog program Π with a clause *C* of the form

$$A \;\mathbf{:-}\; \underline{\boldsymbol{A}}_0, B_1, \ldots, B_n, \underline{\boldsymbol{A}}_1$$

where B_j, $j \geq 1$, are non-recursive subgoals, B_j being a call to procedure P_j in Π, $\underline{\boldsymbol{A}}_0$ and $\underline{\boldsymbol{A}}_1$ are (possibly empty) finite sequences of subgoals not relevant to the analysis, and there is a *schema substitution* Θ associating concrete values with the schema variables such that $G_j\,\Theta = B_j$ and $S_j\,\Theta = P_j$. In Appendix A we describe a means to obtain schema substitutions for our proposed enhanced schemata.

Example 1: Let there be the following program

```
sum([],0).
sum([X|Xs],S):-
       sum(Xs,SXs),
       S is X + SXs.

count([],0).
count([X|Xs],C):-
       count(Xs,CXs),
       C is 1 + CXs.

p(L,S,C):-  sum(L,S),count(L,C).
```

The same list structure is being processed twice in the body of predicate *p/3*, an example of inefficient design choice due to the reuse of existing code. This situation is quite common and can be depicted using program schemata, together with a prescribed way to improve it, all within the following schema-based transformation, **`a1`**:

```
                                    S1([],«&11»):-             S2([],«&21»):-
                                          {process11}                {process21}
                                    S1([X|Xs],«&12»):-         S2([Y|Ys],«&22»):-
                                          {process12}                {process22}
                                          S1(Xs,«&13»),              S2(Ys,«&23»),
〈 ‹S1(L,«&1»),S2(L,«&2»)›, ‹              {process13}      ,         {process23}     ›,

                                          S1_S2_a1([],«&11»,«&21»):-
                                                {process11},
                                                {process21}
                                          S1_S2_a1([X|Xs],«&12»,«&22»):-
                                                {process12},
                                                {process22},
                                                S1_S2_a1(Xs,«&13»,«&23»),
                                                {process13},
        ‹ S1_S2_a1(L,«&1»,«&2»)›, ‹             {process23}                       › 〉
```

Predicate variable symbols **`S1`** and **`S2`** abstract the actual predicate names of the input schemata and provide the name of the new, more efficient, combined predicate. A suffix **`a1`** is attached to the name of the new predicate, indicating the name of the transformation used.

The body of predicate *p/3* matches this transformation, with the following schema substitution Θ,

```
{ S1/sum, «&1»/{S}, S2/count, «&2»/{C},
  «&11»/{0}, {process11}/true, «&21»/{0}, {process21}/true,
  X/X, Xs/Xs, «&12»/{S}, {process12}/true, «&13»/{SXs}, {process13}/{S is X + SXs},
  Y/X, Ys/Xs, «&22»/{C}, {process22}/true, «&23»/{CXs}, {process23}/{C is 1 + CXs}  }
```

This means that the transformation above is successfully matched by the conjunction comprising the body of *p/3*, since the definitions of *sum/2* and *count/2* fit in its input schemata. The following more efficient program can be obtained by the appropriate application of this transformation to the program above:

```
p(L,S,C):-  sum_count_a1(L,S,C).

sum_count_a1([],0,0).
sum_count_a1([X|Xs],S,C):-
      sum_count_a1(Xs,SXs,CXs),
      S is X + SXs,
      C is 1 + CXs.
```

where the list is now processed only once in *p/3* while its elements are being summed and counted at the same time. The schema-based transformation is used by having first the definition of the predicates *sum/2* and *count/2* matched against the input schemata. This matching, if successful, automatically assigns the appropriate values to the abstract constructs of the output schema, thus defining the new predicate *sum_count_a1/3*.

Schema-based transformations greatly reduce and simplify human intervention in the process of program analysis and conversion. User interaction, if needed, is not at a level of complicated unfold/fold rules, but at the level of the structure of the procedure being considered, depicted in a simple fashion by a schema. Sets of transformations can be gradually composed, addressing generic or more specific program constructions.

We propose below a number of enhancements to Gegg-Harrison's schema language, enabling larger classes of programs to be addressed and, hence, more powerful program transformations to be devised.

4 Enhanced Schema-Based Transformations

Simple enhancements can be made to the schema language, giving it more expressiveness. Program transformations can be devised using this enhanced schema-language addressing constructs which are beyond the scope of the work by Fuchs and Fromherz.

Example 2: Let there be the following program

```
sum([],0).
sum([X|Xs],S):-
      sum(Xs,SXs),
      S is X + SXs.

dcount(0,[]).
dcount(C,[X|Xs]):-
      dcount(CXs,Xs),
      C is 1 + CXs.

p(L,S,C):-  sum(L,S),dcount(C,L).
```

which is very similar to the previous example, were it not for the unusual definition of the *dcount/2* predicate to count the elements of a list, with the inversion of the conventional argument order. A more elaborate schema-based program transformation system which accounted for different argument positions would have to be employed if we were to obtain the following combined version of *sum_dcount/3*:

```
sum_dcount(0,[],0).
sum_dcount(C,[X|Xs],S):-
      sum_dcount(CXs,Xs,SXs),
      S is X + SXs,
      C is 1 + CXs.
```

We have enhanced the schema language to allow the reference to argument positions other than the first ones. This enables the program schemata to account for different argument positions sharing the same values, thus allowing program transformations of the form above to be successfully performed. Extra computational costs are, of course, involved in this more expressive proposal and are discussed below.

4.1 Explicit Representation of Argument Positions

A notational device must be employed to enable the proper reference of specific argument positions. We suggest that a number depicting its position be added to each argument. The **schema_A*** shown previously, incorporating this enhancement, is

```
schema_A*(«&1»,[]#N,«&2»).
schema_A*(«&3»,[Head|Tail]#N,«&4»):-
        <pre_process(«&5»,Head#M,«&6»),>
        schema_A*(«&7»,Tail#N,«&8»)
        <,post_process(«&9»,Head#P,«&10»)>.
```

That is, the list argument occupies position **N**, and it is such that its references in the head goal of the first clause and in the head and recursive goals of the second clause are all to the same argument position. The argument positions of variable **Head** in the other subgoals are not subject to any constraint and may thus bear any value.

In order to match actual programs against these enhanced schemata a simple translation must be performed. This translation maps a predicate definition onto an equivalent version in which the argument positions of each subgoal are explicitly represented by the construct "**#N** ", where **N** is the argument position. More formally, the following relationship, describing a generic subgoal and its representation with explicit argument positions holds:

$$p(t_1, \ldots, t_n) \Leftrightarrow p(t_1\#1, \ldots, t_n\#n)$$

A predicate such as *dcount/2*, using the above relationship, would be thus represented as

```
dcount(0#1,[]#2).
dcount(C#1,[X|Xs]#2):-
        dcount(CXs#1,Xs#2),
        C#1 is (1 + CXs)#2.
```

The translation back to its original argument-position-free form must ensure that subgoals employing the pre-defined **is** predicate preserve their original form and that the "**#N**" constructs are not considered as part of the original mathematical expression. In Appendix A we formally describe the mapping between a conventional program and its explicit argument position format.

4.2 Enhanced Program Schemata: Syntax and Semantics

A number of notational shorthands are available during the preparation of schemata. These schemata are generic descriptions of programs and their components are aimed at addressing portions of predicate definitions in an economical manner, allowing irrelevant details to be disregarded and important parts to be brought into focus. The notation should provide us with the necessary elements to allow the description of large classes of constructs sharing particular features, ignoring unimportant variations. The following notational shorthands are offered:

t#*n* – represents the constraint that term *t* occupies position *n* in the subgoal.

{goals*n***}** – represents a possibly empty sequence of *non-recursive* subgoals; we have adopted the more neutral terminology "goals" instead of "process".

{goals**n***}** – represents a possibly empty sequence of *any* sort of subgoals.

&*n* – stands for a possibly empty sequence of arguments.

*x***#1 is 'F'(&***n***,***y***,&***m***)#2** – represents a Prolog **is** subgoal in our explicit argument position format, such that the expression on the right-hand side has at least one reference to the variable symbol *y*. *All* references to *y* are economically represented by a single occurrence of its symbol.

A formal description of the schema language is presented in Appendix A. Within program schemata ordinary Prolog constructs can be used, in which case they are considered as constraints, since they are specific syntactic patterns. Second-order predicate variables are also available so as to enable generic reference to subgoals. One can envisage a spectrum in which larger classes of programs can be depicted with more abstract schemata: at one end of this spectrum, an actual Prolog program is a very specific schema describing those programs with exactly the same form, but possibly different variable names. At the other extreme, a schema can be produced which, for instance, stands for all singly-recursive logic programs:

```
S(&1):-
      {goals1}
S(&2):-
      {goals2},
      S(&3),
      {goals3}
```

The previous program transformation **a1** can be reformulated, using our proposed enhancements, as the following transformation **a1***:

```
⟨ ‹S1(&1,L#N,&2),S2(&3,L#M,&4)›, ‹ S1(&111,[]#N,&112):-         S2(&211,[]#M,&212):-
                                      {goals111}                    {goals211}
                                    S1(&121,[X|Xs]#N,&122):-      S2(&221,[Y|Ys]#M,&222):-
                                      {goals121},                   {goals221},
                                      S1(&123,Xs#N,&124),           S2(&223,Ys#M,&224),
                                      {goals122}                  , {goals222}              ›,

      ‹S1_S2_a1(&1,&3,L#_,&2,&4)›, ‹ S1_S2_a1(&111,&211,[]#_,&112,&212):-
                                       {goals111},
                                       {goals211}
                                     S1_S2_a1(&121,&221,[X|Xs]#_,&122,&222):-
                                       {goals121},
                                       {goals211},
                                       S1_S2_a1(&123,&223,Xs#_,&124,&224),
                                       {goals122},
                                       {goals222}                              › ⟩
```

This transformation covers both previous examples, successfully transforming the programs into a more efficient version. The resulting procedure is such that its argument positions are not important (underscore symbols have been used as argument positions) since they will be translated back into a conventional logic program form without the explicit argument positions. As another example, let there be the following program transformation **a2***:

```
  ⟨ ‹S(&1,L#N,&2,R#M,&3)›, ‹ S(&11,[]#N,&12,I#M,&13):-
                               {goals11}
                             S(&21,[X|Xs]#N,&22,Ac#M,&23):-
                               {goals21},
                               S(&24,Xs#N,&25,Ac1#M,&26),
                               {goals22},
                               Ac#1 is 'F'(&27,Ac1,&28)#2,
                               {goals23}                     ›,

‹S_a2(&1,L#N,&2,I#_,R#_,&3)›, ‹ S_a2(&11,[]#N,&12,NAc#_,NAc#_,&13):-
                                  {goals11}
                                S_a2(&21,[X|Xs]#N,&22,TAc#_,Ac#_,&23):-
                                  {goals21},
                                  NAc#1 is 'F'(&27,TAc,&28)#2,
                                  S_a2(&24,Xs#N,&25,NAc1#_,Ac#_,&26),
                                  {goals22},
                                  {goals23}                             › ⟩
```

It transforms a class of non-tail-recursive programs into more efficient tail-recursive versions, by introducing an accumulator. The version of the *count*/2 predicate of Example 1, with its explicit argument positions, matches the input schema with the following substitution:

```
{S/count, &11/∅,N/1,&12/∅,I/0,M/2,&13/∅,{goals11}/true,&21/∅,&22/∅,Ac/C,&23/∅,
&24/∅,&25/∅,&26/∅,{goals21}/∅,Ac1/CXs,{goals22}/∅,&27/[1,+],&28/∅,{goals23}/∅}
```

These instantiations define the new transformed program, which, after the removal of the empty argument- and subgoal vectors and the explicit argument positions, has the following form

```
count_a2([],NAc,NAc).
count_a2([X|Xs],TAc,C):-
      NAc is 1 + TAc,
      count_a2(Xs,NAc,C).
```

The schema language has been further extended by allowing the abstract description of functors and their arguments and the abstract description of constants and system predicates which serve as tests. Constants are generically denoted as **ct(*n*)** where *n* works as an index to differentiate among distinct constants in the same schema. Prolog tests, viz. the arithmetic comparators **>**, **<**, **>=**, etc., and *integer/1*, *ground/1*, *number/1*, etc., are generically denoted by **'T*n*'** where *n* works as an index and may be omitted. Using these additional shorthands even larger classes of programs can be depicted by simple schemata. For instance, the schema below depicts the class of programs dealing with data structures in a singly-recursive fashion, such that its termination occurs when a constant is found or when a special condition is met by part of the data structure:

```
S(&11,ct(_)#N,&12):-
      {goals11}
S(&21,F(&22,X,&23)#N,&24):-
      {goals21}
      'T'(&25,X#_,&26),
      {goals22}
S(&31,F(&32,Xs,&33)#N,&34):-
      {goals31}
      S(&35,Xs#N,&36),
      {goals32}
```

We have also incorporated in our schema language a facility to unify two terms: the construct **unify**(t_1, t_2) unifies the terms t_1 and t_2 within the clause in which they appear. This construct is important to program transformations in which explicit instantiations are removed or the unfolding of a procedure takes place. For instance, the following schema-based transformation appropriately unfolds a procedure with a single clause onto its invocation:

$$\left\langle\ \langle \texttt{S(\&1)} \rangle,\ \left\langle \boxed{\begin{array}{l}\texttt{S(\&2):-}\\ \quad\quad \texttt{\{goals\}}\end{array}} \right\rangle,\ \langle \texttt{unify(\&1,\&2),\{goals\}} \rangle,\ \varnothing\ \right\rangle$$

that is, a call **S(&1)** to procedure **S** can be replaced by the body **{goals}** of the single clause comprising **S**, once the values of the arguments **&1** of the call have been unified with those arguments **&2** in the head goal of procedure **S**. A repertoire of schema-based program transformations is shown in Appendix B. These are the transformations currently available in our implemented system, but other components can be added on to them.

4.3 Computational Complexity of Enhanced Program Schemata

The feasibility of our proposal to employ enhanced schema-based transformations depends on the computational complexity involved in matching an actual program to a schema. It turns out that this complexity is a polynomial function on a number of parameters, such as the number of variables in the program, the number of schema variables and so on.

Let there be a procedure P in its explicit argument position version, comprised of clauses $C_1, \ldots, C_n$ each of which of the general form

$$\begin{array}{l} p(x_1\#1,\ldots,\ x_r\#r)\texttt{:-}\\ \quad\quad p_1(x_{[1,\,1]}\#1,\ldots,\ x_{[r_1,\,1]}\#r_1),\\ \quad\quad\quad \vdots\\ \quad\quad p_s(x_{[1,\,s]}\#1,\ldots,\ x_{[r_s,\,s]}\#r_s). \end{array}$$

where p and p_i are generic predicate symbols (recursive or not). Let us also assume the existence of a schema S with clauses $D_1, \ldots, D_n$ each of which of the general form

$$
\begin{array}{l}
Q(\&v_1,\ y_1\#z_1,\ \&v_2,\ldots,\ \&v_t,\ y_t\#z_t,\ \&v_{t+1}) \texttt{:-} \\
\quad \texttt{\{goals}_1\texttt{\}}, \\
\quad Q_1(\&v_{[1,1]},\ y_{[1,1]}\#z_{[1,1]},\ \&v_{[2,1]},\ldots,\ \&v_{[t_1,1]},\ y_{[t_1,1]}\#z_{[t_1,1]},\ \&v_{[t_1+1,1]}), \\
\quad \texttt{\{goals}_2\texttt{\}}, \\
\quad\quad \vdots \\
\quad \texttt{\{goals}_u\texttt{\}}, \\
\quad Q_u(\&v_{[1,u]},\ y_{[1,u]}\#z_{[1,u]},\ \&v_{[2,u]},\ldots,\ \&v_{[t_1,u]},\ y_{[t_u,u]}\#z_{[t_u,u]},\ \&v_{[t_u+1,u]}), \\
\quad \texttt{\{goals}_{u+1}\texttt{\}}.
\end{array}
$$

where Q and Q_i are generic predicate variable symbols (recursive or not). The number of possible matches between an actual subgoal in a clause C_i of the program and a clause D_i of the schema is a function of the variables and slots contained in each construct. For instance, for a schema subgoal with two variable slots and three schema variables,

$$Q(\&v_1,\ y_1\#z_1,\ \&v_2,\ y_2\#z_2,\ \&v_3)$$

we have that a subgoal $p(x_1\#1,\ x_2\#2)$ with two variables has only one possible matching,

Q	$\&v_1$	$y_1\#z_1$	$\&v_2$	$y_2\#z_2$	$\&v_3$
p	$\varnothing$	$x_1\#1$	$\varnothing$	$x_2\#2$	$\varnothing$

whereas $p(x_1\#1,\ x_2\#2,\ x_3\#3)$ with three variables has, in its turn, three different possibilities,

Q	$\&v_1$	$y_1\#z_1$	$\&v_2$	$y_2\#z_2$	$\&v_3$
p	$\varnothing$	$x_1\#1$	$\varnothing$	$x_2\#2$	$x_3\#3$
p	$\varnothing$	$x_1\#1$	$x_2\#2$	$x_3\#3$	$\varnothing$
p	$x_1\#1$	$x_2\#2$	$\varnothing$	$x_3\#3$	$\varnothing$

The number of possible matchings between the concrete subgoal p_i and the schema subgoal Q_j, denoted by $m_{[p_i,\,Q_j]}$, corresponds to the different possibilities of fitting the existing variables of p_i into the variable slots of Q_j. This can be depicted as the summation

$$m_{[p_i,\,Q_j]} = \sum_{i=1}^{r-t+1} i = (r - t + 1)\,(r - t + 2)/2 = (r^2 + t^2 - 2\,rt + 3r - 3t + 2)/2$$

where r and t are respectively the number of variable symbols of p_i and the number of variable slots of Q_j. The $\texttt{\{goals}_m\texttt{\}}$ slots also contribute to the number of possible matchings of a clause C_i in P and a clause D_i in S and must be taken into account: in a rather similar fashion as the previous equation, their possible matchings m_{goals} can be summarised by the equation

$$m_{goals} = \sum_{i=1}^{(u+1)-s+1} i = ((u+1) - s + 1)\,((u+1) - s + 2)/2 = (u^2 + s^2 - 2us + 5u - 5s + 6)/2$$

where $u + 1$ is the number of $\texttt{\{goals}_m\texttt{\}}$ slots of D_i and s is the number of subgoals in C_i. If we assume that all the predicate and schema clauses have the same number of subgoals and subgoal slots, respectively s and $u + 1$, and that their subgoals have the same number r and t of variable symbols and variable slots in each clause, then we have the situation that the number of possible matchings between one clause C_i of P and one clause D_i of S, $m_{[C_i,\,D_i]}$, is the product of the matchings $m_{[p_i,\,Q_i]}$ of each subgoal times the number $s + 1$ of subgoals in P, times the number of possible matchings between the subgoals in P and subgoals slots in S, that is,

$$m_{[C,\,D]} = m_{[p_i,\,Q_i]}\,(s+1)\,m_{goals}$$

If we replace the equations above in it we have

$$m_{[C,\,D]} = (r^2 + t^2 - 2\,rt + 3r - 3t + 2)\,(s+1)\,(u^2 + s^2 - 2us + 5u - 5s + 6)/4$$

Finally, the total matchings $m_{[P,\,S]}$ between P and S would also have to account for the number of clauses n of P and S, that is,

$$m_{[P,\,S]} = n\, m_{[C,\,D]} = n\,(r^2 + t^2 - 2\,rt + 3r - 3t + 2)\,(s + 1)\,(u^2 + s^2 - 2us + 5u - 5s + 6)/4$$

This final equation is such that the number of possible matchings between a given program and a schema is of polynomial magnitude: if we choose any of the variables above as the focus of a numerical analysis and assign arbitrarily high values to the other variables, a polynomial equation will always be obtained.

The interesting feature of the enhanced schema language that is responsible for their polynomial (and not exponential) number of matchings is that permutations of constructs cannot be expressed. This might seem a negative property in that the expressiveness of our schema language does not account for simple changes in the ordering of arguments or subgoals, but this problem can easily be circumvented by adding different schemata for different orderings.

The analysis above has been carried out considering the worst case of the matching process, in which no syntactic constraints have been imposed and only predicate variables and subgoal slots are found in the schema. In practical applications, however, general constructs are interspersed with specific syntactic constraints thus reducing the number of possible matchings to be considered.

In our approach, only the *first* successful match between a schema and a procedure is actually used in the transformation process. If there is a constraint in the schema such that every partial match has to be tried before the final complete match is obtained, then this will require a polynomial number of attempts.

5 An Opportunistic Approach for Improving the Efficiency of Prolog Programs

Our suggested approach consists of analysing a given program Π, searching for opportunities to employ the repertoire of schema-based transformations. This is a realistic scenario in which a program is analysed with respect to a set of transformations suggesting improvements to common inefficient Prolog constructs. There are two possible ways to use our transformations:

- By checking if a *procedure* $P \subseteq \Pi$ can have a transformation applied to it; or
- By checking if a *conjunction* $G_1, \ldots, G_n$ appearing in the body of some clause $C \in \Pi$ can have a transformation applied to it.

Both alternatives are pursued during the analysis of the program. An extended program Π' is eventually obtained such that if $\Pi \Rightarrow G$ then $\Pi' \Rightarrow G$, where "$\Rightarrow$" is the logical implication symbol. We can say that Π' *subsumes* Π in the sense that the answers to queries posed in the latter are still obtainable in the former; however, since new procedures are introduced in Π', some queries can be posed to Π' (and their answers would be appropriately obtained) which do not apply to Π.

5.1 Transforming Isolated Procedures

An isolated procedure P is analysed for possible improvements by being checked against the input schema S of some transformation T. Obviously, only those transformations with a single input schema should be considered here.

Let $\mathrm{T} = \langle\langle G\rangle, \langle S\rangle, \langle H\rangle, \langle T\rangle\rangle$ be an enhanced schema-based transformation such that $S\Theta = P$, for some procedure P, $P \subseteq \Pi$, and P', $T\Theta = P'$, the recommended transformed version of P. This can be understood as the transformation suggesting improvements to be made to those procedures P matching the input schema S, or, alternatively, as a suggestion for the replacement of those procedures P by the procedure P' obtained after applying the substitution Θ to the output schema. However, in order to carry out this replacement and properly extend Π, an additional clause must be introduced making the connection between the old procedure P to be removed and the newly obtained procedure P' to be inserted. This additional clause will guarantee that queries posed to the new program Π' concerning the removed procedure P be appropriately translated into queries concerning P'.

Let I be the head goal of the clauses comprising P with fresh, uninstantiated variables, and I' the head goal of those clauses comprising P' also with fresh, uninstantiated variables. The *conversion clause* $C_{[P,\, P']}$ between P and P', of the form I **:-** I', is such that $G\Theta^* = I$, and $H\Theta^* = I'$ for some extension Θ^* of substitution Θ. The final transformed program Π' can be obtained as the result of the following set operation

$$\Pi' = (\Pi - P) \cup P' \cup \{C_{[P,\, P']}\}$$

Conversion clauses can be seen as *initialisation calls* [O'Keefe 90; Sterling & Shapiro 94] in which values are initially assigned to argument positions of a predicate being invoked.

Example 3: Let there be a program Π consisting solely of procedure *count/2*, shown in example 1 above. We have shown in Subsection 4.2 that transformation **a2*** can successfully be used to obtain a more efficient procedure *count_a2/3*. In order to appropriately alter Π, removing the inefficient procedure *count/2* and inserting the new, more efficient procedure *count_a2/3*, we would also have to incorporate the following conversion clause

```
count(L,C):-
      count_a2(L,0,C).
```

This conversion clause converts those queries previously answered via *count/2* to queries now answered via *count_a2/3*. The new program Π' would thus be

```
count(L,C):-
      count_a2(L,0,C).
count_a2([],NAc,NAc).
count_a2([X|Xs],TAc,C):-
      NAc is 1 + TAc,
      count_a2(Xs,NAc,C).
```

This new program subsumes the original one: the answers to queries posed to the initial *count/2* procedure will still get the same answers, and additional queries to *count_a2/3* procedure can now be posed. This newly obtained procedure can be further combined, as we shall see in the example below.

5.2 Transforming Programs by Combining Procedures

As shown in our example 1 above, consecutive subgoals within the body of a clause may also provide a prospective opportunity for improvement of the performance of a program. In this case, a given program Π has its conjunctions, i.e. sequences of more than one subgoal in the body of one of its clauses, analysed by checking them and their respective procedures against the schema patterns of our transformations. Both calling patterns and procedures are used in this analysis.

The scenario for this analysis is a program Π and one of its clauses C, of the form

$$A \textbf{ :- } \underline{\boldsymbol{A}}_0, B_1, \ldots, B_n, \underline{\boldsymbol{A}}_1$$

where B_j, $j \geq 1$, are non-recursive subgoals, B_j being a call to procedure $P_j \subset \Pi$, and $\underline{\boldsymbol{A}}_0$ and $\underline{\boldsymbol{A}}_1$ are (possibly empty) finite sequences of subgoals not relevant to the analysis. If there is a schema-based transformation $\mathrm{T} = \langle \langle G_1,\ldots,G_n\rangle, \langle S_1,\ldots,S_n\rangle, \langle H_1,\ldots,H_m\rangle, \langle T_1,\ldots,T_m\rangle \rangle$ and a substitution Θ such that $G_j\,\Theta = B_j$ and $S_j\,\Theta = P_j$, $1 \leq j \leq n$, then a *transformed clause* C' is obtained, of the form

$$A \textbf{ :- } \underline{\boldsymbol{A}}_0, B'_1, \ldots, B'_m, \underline{\boldsymbol{A}}_1$$

where $H_k\,\Theta = B'_k$, $1 \leq k \leq m$. The transformed clause incorporates the changes suggested by the transformation, invoking new procedures P'_k, such that $T_k\Theta = P'_k$, $1 \leq k \leq m$. The new transformed program incorporating these suggestions can be obtained by replacing clause C by its transformed version C' and inserting the new procedures P'_k. This can be described by the following set operation

$$\Pi' = (\Pi - \{C\}) \cup \{C'\} \cup P'_1 \cup \ldots \cup P'_m$$

The procedures P_j, $1 \le j \le n$, previously employed by C are not altered since they may be used by other procedures and also because we are aiming at an extension of a program, which means that everything provable in Π, including queries to P_j, $1 \le j \le n$, must also be provable in Π'.

5.3 Program Transformation: Overview

The specific manners of dealing with isolated procedures and conjunctions are put together in the same framework to carry out opportunistic transformations in a given program. The following simple Prolog code describes the manner in which our system has been implemented:

```
transform(P,FP):-
      transform_predicate(P,TP),
      transform(TP,FP).
transform(P,FP):-
      transform_conjunction(P,TP),
      transform(TP,FP).
transform(P,P).
```

Initially, attempts are made at transforming isolated procedures: *transform_predicate/2* transforms one isolated procedure, obtaining a new version which replaces the chosen procedure in the program **P**, together with its conversion clause. The newly obtained program **TP** is then recursively used in further transformations. The second clause, if successful, chooses and transforms by means of *transform_conjunction/2* a conjunction in **P**, inserting a new combined procedure and appropriately replacing the occurrences of the chosen conjunction. The new transformed program **TP** is recursively used in other transformations. When there are no more predicates or conjunctions to be transformed, then we can say that the program **P** has reached its optimal state with respect to the repertoire of transformations of our system and no more improvements can be done – the third clause caters for this possibility.

The order of clauses in the Prolog code above is such that initially all isolated procedures will be tested for prospective transformations. When there are no more procedures that can be isolatedly transformed (*transform_predicate/2* fails), then their combinations, stated in the conjunctions of the program, will be analysed for possible transformations using the method explained above. The resulting program, after a new combined procedure is obtained and the conjunctions are altered, is tested again for any isolated predicate transformations. This process goes on until no more program transformations can be applied.

The ordering of our analysis, considering first isolated procedures and then their combinations, is based on the sort of alterations recommended by each program transformation of our repertoire. Those changes in isolated procedures recommended by our transformations involve the reordering of subgoals and the insertion of extra arguments: features like the relative ordering of arguments and the patterns of recursive data structures are not altered and these play an important role during the combination of procedures. In some cases, as we shall see below, this ordering is immaterial since the same final program (possibly with different names of predicates and variables, but similar structure) can be reached in any event.

Our method incorporates an efficiency-related improvement to the transformation process. Upon the transformation of a conjunction, after the new procedures $P'_1 \cup \ldots \cup P'_m$ are inserted and C is replaced by C', the newly obtained program Π' is analysed and all further occurrences of conjunctions $B_1,\ldots,B_n$ in its clauses are appropriately replaced by $B'_1,\ldots,B'_m$. More formally, each conjunction $D_1,\ldots,D_n$ in Π' such that there is a term substitution θ unifying D_i and B_i, that is, $D_i\theta = B_i$, is replaced by $B'_1\theta,\ldots, B'_m\theta$. This improvement avoids the matching of the procedures being invoked by each subgoal of the conjunction against the schemata of the program transformations. Additionally, whenever an isolated procedure P is transformed into a new procedure P' our method translates any other existing calls to the old procedure P in the new program Π' into appropriate calls to the new procedure *P*'. This allows further examinations and possible transformations of the new procedure, without referring to conversion clauses, which would remain unchanged.

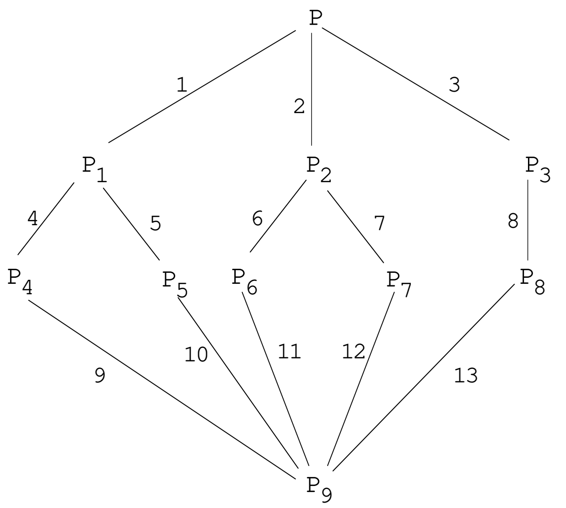
The termination of the process above is only possible if the transformed program eventually converges to a form in which no more transformations can be further applied to it. This convergence depends on the nature of the syntactic alterations prescribed by the available transformations. Each schema-based transformation should be designed bearing in mind the iterative framework within which they will be employed. In our repertoire shown in Appendix B, isolated procedures are altered by changing the order or removing some of its subgoals, and care was taken to ensure that each of them or their interaction would not lead to loops; conjunctions are always replaced by other smaller conjunctions, guaranteeing an eventual convergence of the program to a form in which no other conjunction-altering transformations can be applied.

Example 4: Let us consider again our example 1 and suppose we have only the transformations **`a1*`** and **`a2*`** shown before. The *transform/2* predicate would eventually supply us with the following transformed (extended) program:

```
sum(L,S):-
       sum_a2(L,0,S).

count(L,C):-
       count_a2(L,0,C).

p(L,S,C):-  sum_a2_count_a2_a1(L,0,S,0,C).

sum_a2([],NAc,NAc).
sum_a2([X|Xs],TAc,Ac):-
       NAc is X + TAc,
       sum_a2(Xs,NAc,Ac).

count_a2([],NAc,NAc).
count_a2([X|Xs],TAc,Ac):-
       NAc is 1 + TAc,
       count_a2(Xs,NAc,Ac).

sum_a2_count_a2_a1([],NAc1,NAc1,NAc2,NAc2).
sum_a2_count_a2_a1([X|Xs],TAc1,Ac1,TAc2,Ac2):-
       NAc1 is X + TAc1,
       NAc2 is 1 + TAc2,
       sum_a2_count_a2_a1(Xs,NAc1,Ac1,NAc2,Ac2).
```

The names of the variables have been appropriately changed so as to make the predicates easier to understand. This example is such that if only **`a1*`** and **`a2*`** are available then the ordering of their application is immaterial, for the same final program is eventually obtained, with different predicate and variable symbol names. In Figure 1 below we show the different paths through which the same final program P_9 would be obtained if different orderings of the application of transformations were pursued. On the right-hand side we have listed the labels of the edges connecting the nodes in the diagram. These labels show the pair ‹*P*, *T*› with the chosen procedure or conjunction and the transformation employed. The predicate name, however, would be different in each case, since the transformations compose the new name by using the original predicate symbol(s) of the conjunction and adding a suffix to it.

This phenomenon does not always take place: it depends on the program and available transformations. One may conceive a program and a set of transformations such that the order of their application would play a significant role, some transformations altering the program in such a way that the use of other transformations would be prevented. In the following section we describe our adopted approach to cope with this problem.




```
 1. ‹sum/2,a2›
 2. ‹count/2,a2›
 3. ‹[sum(L,S),count(L,C)],a1›
 4. ‹count/2,a2›
 5. ‹[sum_a2(L,0,S),count(L,C)],a1›
 6. ‹[sum(L,S),count_a2(L,0,C)],a1›
 7. ‹sum/2,a2›
 8. ‹sum_count_a1/3,a2›
 9. ‹[sum_a2(L,0,S),count_a2(L,0,C)],a1›
10. ‹sum_a2_count_a2/4,a2›
11. ‹sum_count_a2_a1/4,a2›
12. ‹[sum_a2(L,0,S),count_a2(L,0,C)],a1›
13. ‹sum_count_a1_a2/4,a2›
```

Figure 1: Different Orderings of the Transformations of `P`

6 Coping with More than one Applicable Transformation: Heuristics

For a given predicate or conjunction there might be more than one applicable transformation at one time. In order to solve this problem, Fuchs and Fromherz [Fuchs & Fromherz 91] adopt the intervention of a human user who chooses one of the prospective transformations and has it applied to the program. Since our system is aimed at analysing full programs and these can be of any size, asking a human user for assistance may not be a practical thing to do. Fuchs and Fromherz, however, suggest that some form of heuristic could be incorporated into each transformation and be used as a means to automatically select the best candidate from a set of prospective schema-based program transformations. We have pursued this suggestion and incorporated simple heuristics to our enhanced schema-based program transformations.

The enhanced schema language and the realistic context in which to use it are more important than the specific program transformations shown here: these should be seen as examples and are not meant to be authoritative nor exhaustive. Following this line, we find of more importance not the particular heuristics assigned to our transformations, but the rationale used to do so. In this section we describe the method we used to assign heuristics to our program transformations: the purpose of this method is to aid humans preparing their own program transformations.

Our approach establishes that opportunities to apply program transformations are continuously sought after. At one moment a given program may have more than one transformation applicable to one of its procedures or conjunctions – these transformations can be sequentially applied if they preserve the overall structure of the program, maintaining argument positions and their relative ordering, number of clauses, and so on. For instance, when the transformation system attempts to obtain a tail-recursive version of the program

```
sum_rev([],0,[]).
sum_rev([X|Xs],S,Rs):-
      sum_rev(Xs,SXs,RXs),
      S is SXs + X,
      append(RXs,[X],Rs).
```

it uses transformation **`p41`** to address the arithmetic expression and transformation **`p43`** to cope with the *append*/3 subgoal (both transformations are shown in Appendix B); thus obtaining the following more efficient program

```
sum_rev(L,S,RL):-
      sum_rev_p4_p4(L,0,S,[],RL).
```

```
sum_rev_p4_p4([],S,S,Rs,Rs).
sum_rev_p4_p4([X|Xs],Ac,S,TRs,Rs):-
      NAc is Ac + X,
      append([X],TRs,NTRs),
      sum_rev_p4_p4(Xs,NAc,S,NTRs,Rs).
```

The ordering of the application of these program transformations is not important, since they do not interfere with each other: in fact they *complement* each other. In this example, the same final program would be achieved in any event. Transformations aimed at procedure combinations behave similarly. The following predicate, for instance,

```
sum_count_rev(L,S,C,RL):-
      sum(L,S),
      dcount(C,L),
      rev(L,RL).
```

where *rev/2* is defined as

```
rev([],Rs).
rev([X|Xs],Rs):-
      rev(Xs,RXs),
      append(RXs,[X],Rs).
```

and *sum/2* and *dcount/2* are defined as in previous examples, would trigger transformation **`a1*`** twice, composing a procedure traversing a list only once while simultaneously summing and counting its elements and preparing the reverse list. Each procedure *sum/2*, *dcount/2* and *rev/2* would, in their turn, trigger isolated procedure transformations, the ordering of which is not relevant, for the transformed program eventually converges to the same form, with possibly different variable and predicate symbols.

We suggest that transformations be assessed for their potential sequential use, maximising the number of transformations applicable to a given initial program. Our underlying assumption is that the more transformations a program has applied to it, the more efficient it should become.

The heuristics should allow the application of the largest number of transformations, delaying the use of those transformations that would prevent others from being employed. This interference is due to the extent and kind of the syntactic alterations performed by these transformations causing substantial changes to a procedure and preventing it from matching the schemata of other transformations. Heuristics should then be assigned to program transformations based on their interchangeability: the less dramatically a transformation alters a program, the best its heuristic should be, since other subsequent transformations will not be prevented from being used. We suggest a set of guidelines to analyse the syntactic changes of programs transformations and establish our heuristics:

- Isolated procedure transformations which only alter the internal parts of a procedure, by changing the order of its subgoals (*e.g.* transf. **`p01`**, in Appendix B), replacing subgoals (*e.g.* transf. **`p21`**) or removing spurious explicit instantiations (*e.g.* transf. **`p31`**) should be the first transformations to be applied. Their syntactic changes are localised and do not impair any further transformations in the resulting procedure.
- Isolated procedure transformations that insert extra arguments but maintain their relative ordering, as in the original procedure, and do not insert new clauses in the procedure nor change the ordering of the clauses should come next. The calls to the changed procedure are accordingly adapted, with the original arguments in their relative ordering extended by the additional inserted parameters: given that in our transformations only the relative ordering is used as a constraint, these changes do not prevent any further transformations from being applied.
- The program transformations which insert new clauses, remove arguments or alter the relative argument ordering should be given the lowest marks since they may prevent further transformations from taking place.

An interesting alternative to these heuristics is to enable our system to pursue all possible orderings in the application of transformations and select the longest path among them. Another altogether different approach would be to devise an evaluation function assessing the transformations carried out by judging the gain in efficiency in the resulting program. Such a function would assign values to the resulting programs, allowing the comparison of alternative options.

7 Conclusions and Directions of Research

Our work extends previous research done by Fuchs and Fromherz [Fuchs & Fromherz 91] on using program schemata to guide program transformation: a schema-based transformation is a generic description of an improvement that can be done to a program. We have proposed a simple extension to the schema language of Gegg-Harrison [Gegg-Harrison 89; Gegg-Harrison 91] which allows arbitrary argument positions to be conveniently addressed: Gegg-Harrison's schemata only represent the first arguments of a goal. This enhancement has been incorporated to the schema-based program transformations of Fuchs and Fromherz thus allowing a larger class of programs to be addressed with the same schema-based transformation and be successfully transformed.

We have suggested a realistic context in which enhanced schema-based transformations can be used to perform actual program analysis. An opportunistic approach to program analysis and transformation has been implemented by means of which a given Prolog program has its constructs checked against the repertoire of program transformations: these are applied as long as there are opportunities to do so. The repertoire of program transformations have a decisive effect on the termination of this process: their syntactic modifications should ensure that the resulting program differs substantially from its preceding versions so as not to trigger any subsequence of those transformations applied so far. This constraint is, of course, added on to those essential restrictions that program transformations should preserve the meaning (*i.e.* set of logical consequences) and the termination status of the original program and confer to it a better execution performance.

In this paper we have also discussed the computational costs of matching an enhanced program schema and an actual Prolog program – this can be stated as a polynomial equation involving the number of clauses and subgoals of both program and schema and the number of variables of the program and schema variables of the schema. Exponential combinations are avoided due to the schema language not allowing permutations of components to be expressed. Different orderings can be achieved with different schemata, though.

We have introduced some useful enhanced schema-based program transformations which, once applied to a program, guarantee an improvement in its performance. It must be pointed out that our repertoire is by no means complete or exhaustive and providing such complete set of transformations is not our intention. Our main contributions are a language with which one can define program transformations and a context in which this should be used: these should be seen as tools that experienced programmers can use to devise their own strategies for program transformation and to make them available to the users of their program transformation systems.

We have additionally investigated the suggestion of Fuchs and Fromherz to associate heuristics with each transformation and use it as a means to automatically select the best candidate from a set of prospective schema-based program transformations. Since our intention is to encourage programmers to devise their own schema-based transformations, we have suggested guidelines to help them establish their own heuristics. These guidelines aim at maximising the number of transformations applicable to a given initial program, and their underlying assumption is that the more transformations a program has applied to it, the more efficient it should become.

A prototypical implementation of our approach has been devised using LPA MacProlog32, reusing large adapted portions of the system by Fuchs and Fromherz. The available schema-based transformations are those shown in Appendix B. All those transformations available in the system of Fuchs and Fromherz can be translated into our proposed extended language in a straightforward way, or they can be extended to cover a larger number of constructs, for instance, generalising functors and constants, inserting schema variables, and so on. Our repertoire is limited to singly-recursive procedures with two clauses, but as mentioned before, other special-purpose or more sophisticated transformations can be further added on to it.

In order to aid the preparation of schema-based program transformations, two useful extensions to our research would be a more user-friendly visual representation for schemata and an editor, along the lines of the work described in [Vasconcelos 93], incorporating it. Schema variables and slots would be represented in a graphic form which would differentiate them from actual Prolog constructs. An initial attempt at such visual language representing schemata which associates graphic elements with the schema slots, thus improving the visualisation of the schema, is illustrated by the example in Figure 2 below, where the numbered squares and rectangles stand for schema variables and subgoal slots, respectively.

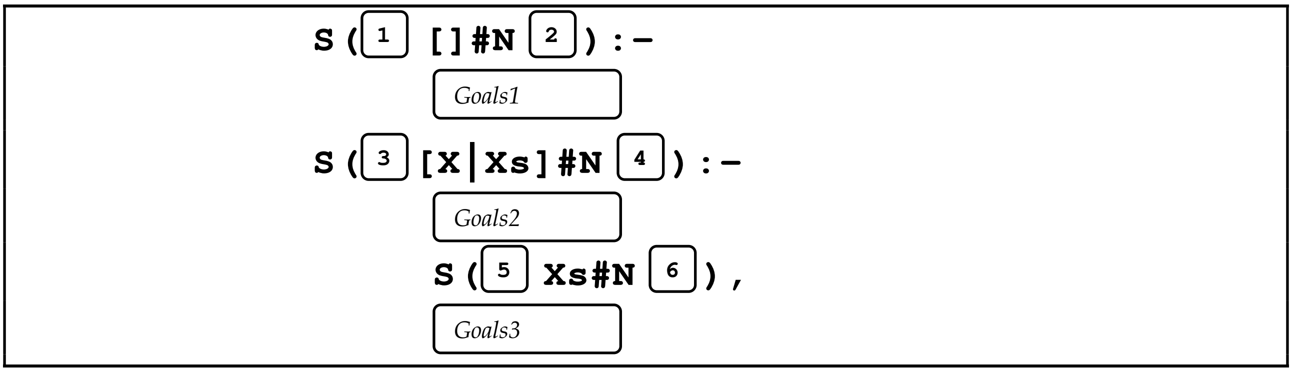


Figure 2: Graphic Representation of a List-Processing Schema

Enhanced schemata can also be used as templates to aid program construction. The programming techniques mentioned in our introductory section can be economically depicted by means of such constructs: schemata can be offered to users of techniques editors such as those described in [Bowles et al. 94] and [Robertson 91] and also be explicitly used to introduce programming techniques in Prolog courses. Programming techniques extracted from actual programs [Vasconcelos 94] can be conveniently represented as schemata and used in techniques editors.

Non-syntactic additional constraints are sometimes necessary to properly describe a schema. It is not possible with our enhanced schema language to specify that a certain variable should or should not be found in a vector of goals or in a vector of arguments. Another interesting example of additional non-syntactic constraint is to state that a given vector of goals or arguments should or should not be empty. We suggest that a simple set-theoretical language be employed to cope with these issues, the schemata language being further augmented to accommodate it. According to this suggestion, schemata would consist of a syntactic part followed by a sequence of additional constraints using this set-theoretical language. For instance, the following augmented schema could be represented:

```
S(&11,[]#N,&12):-
  {goals11}
S(&21,[X|Xs]#N,&22):-
  {goals21},
  S(&23,Xs#N,&24),
  {goals22}
```

```
{goals21} ≠ ∅  ∧  X ∉ {goals21} ∧  &22 = ∅
```

The syntactic description of the procedure in the schema language remains unaltered, as shown in the upper rectangle, but it is complemented by a list of non-syntactic constraints, which in the example above consists of further specifying **`{goals21}`** as being a non-empty vector such that the variable symbol **`X`** is found in one of its subgoals and that the vector of arguments **`&22`** is not empty.

A useful alternative framework for program development would combine programming techniques and program transformations. The use of programming techniques may provide us with programs whose individual procedures are optimally efficient but their overall behaviour (*i.e.* inter-procedural relationships) can be improved. Subsequent program transformations could provide us with the desired overall efficiency. For instance, procedures *sum*/2 and *count*/2 of Example 1 could have been prepared by means of ready-made programming techniques, but their combined use in procedure *p*/3 would still be an inefficient programming construct easily detectable, as our example showed, by the program transformation system.

Acknowledgements: Thanks are due to R. Schwitter and S. Simpson for proof-reading earlier versions of this paper and providing useful suggestions. The first author can also be reached at **wamb@aisb.ed.ac.uk** or **wamb@fortal.uece.br**.

References

[Bowles et al. 94] A. Bowles, D. Robertson, W. Vasconcelos, M. Vargas-Vera, D. Bental, Applying Prolog Programming Techniques, in: International Journal of Human-Computer Studies, Vol. 3, pp. 329–350, 1994.

[Clocksin & Mellish 87] W. F. Clocksin, C. S. Mellish, Programming in Prolog (3rd Edition), Springer, 1987.

[Deville 90] Y. Deville, Logic Programming: Systematic Program Development, Addison-Wesley Publishing Company, 1990.

[Fuchs & Fromherz 91] N. E. Fuchs, M. P. J. Fromherz, Schema-Based Transformations of Logic Programs, in: K.-K. Lau, T. Clement (Eds.), Proceedings of LOPSTR '91, University of Manchester, 1991, Springer, 1992.

[Gegg-Harrison 89] T. S. Gegg-Harrison, Basic Prolog Schemata, Report CS-1989-20, Department of Computer Science, Duke University, 1989.

[Gegg-Harrison 91] T. S. Gegg-Harrison, Learning Prolog in a schema-based environment, Instructional Science, 20, pp. 173-192, 1991.

[Kirschenbaum et al. 94] M. Kirschenbaum, S. Michaylov, L. Sterling, Skeletons and Techniques as a Normative Approach to Program Development in Logic Languages, Report CES-94-08, Case Western Reserve University, 1994.

[Lakhotia & Sterling 90] A. Lakhotia, L. Sterling, How to Control Unfolding when Specialising Interpreters, in: L. Sterling (Ed.), The Practice of Prolog, MIT Press, pp. 171 ff., 1990.

[Lloyd 93] J. W. Lloyd, Foundations of Logic Programming, Second Extended ed., Springer, 1993.

[Nielson & Nielson 90] H. R. Nielson, F. Nielson, Eureka Definitions for Free – Disagreement Points for Fold/Unfold Transformations, Proceedings ESOP'90, LNCS 432, Springer-Verlag, 1990.

[O'Keefe 90] R. A. O'Keefe, The Craft of Prolog, MIT Press, 1990.

[Pereira & Shieber 87] F. C. N. Pereira, S. M. Shieber, Prolog and Natural Language Analysis, CSLI Lectures Notes 10, 1987.

[Proietti & Pettorossi 90] M. Proietti, A. Pettorossi, Synthesis of Eureka Predicates for Developing Logic Programs, Proceedings ESOP'90, LNCS 432, Springer-Verlag, 1990.

[Robertson 91] D. Robertson, A Simple Prolog Techniques Editor for Novice Users, in: Proceedings of 3rd Annual Conference on Logic Programming, Edinburgh, Scotland, 1991.

[Ross 89] P. Ross, Advanced Prolog, Addison-Wesley, 1989.

[Sterling & Kirschenbaum 91] L. Sterling, M. Kirschenbaum, Applying Techniques to Skeletons, Report TR 91-179, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1991.

[Sterling & Shapiro 94] L. Sterling, E. Y. Shapiro, The Art of Prolog, Advanced Programming Techniques, Second Edition ed., MIT Press, 1994.

[Tamaki & Sato 84] H. Tamaki, T. Sato, Unfold/Fold Transformation of Logic Programs, University of Uppsala, 1984.

[Vasconcelos 93] W. W. Vasconcelos, Designing Prolog Programming Techniques, in: Proceedings of 3rd International Workshop on Logic Program Sythesis and Transformation (LoPSTr '93), Louvain-La-Neuve, Belgium, Springer-Verlag, 1994.

[Vasconcelos 94] W. W. Vasconcelos, Extracting Prolog Programming Techniques, in: Proceedings of 11th Brazilian Symposium on Artificial Intelligence, Fortaleza, Ceará, Brazil, Also available as Research Paper 715, Department of Artificial Intelligence, University of Edinburgh, Scotland, UK, 1994.

Appendix A: Syntax and Semantics of the Enhanced Schema Language

In this appendix we formally introduce the schema language to depict logic programs. This language is based on Gegg-Harrison's schemata [Gegg-Harrison 89; Gegg-Harrison 91] extending it to represent any argument position; other enhancements have also been incorporated such as second-order variables to generalise functors and arithmetic expressions. In order to describe our schema language, we employ two kinds of fonts: the symbols in *italics* are meta-variables, *i.e.* they stand for generic constructs whose details are further explained; and the symbols in **`type-writer font`** are the actual constructs of our language. We shall use the sequence notation $\langle C_1, \ldots, C_n \rangle$ to refer to collections of objects whose ordering is relevant: by doing so we avoid unimportant implementational details or unnecessary commitments to data structures. The empty sequence will be represented by "$\langle\,\rangle$" and the operator "$\cdot$" concatenates two sub-sequences.

A.1 Enhanced Schemata Language

An *enhanced schema* **S** is a sequence of *schematic Horn clauses* $C_1, \ldots, C_n$ such that each C_i is of the form H **`:-`** $G_1, \ldots, G_m$. The components $H, G_1, \ldots, G_m$ are the *schematic atoms* of C_i, H being its *schematic head* and $G_1, \ldots, G_m$ its *schematic body*.

A *schematic atom* G is either

- a *first-order schematic atom* p**`(`**$T_1, \ldots, T_n$**`)`** where p is a predicate symbol and $T_1, \ldots, T_n$ are *schematic terms* in their explicit argument position form. Predicate symbols are Prolog atoms, that is, strings starting with *non-capital* letters, followed by any number of letters, digits and underscore symbols;
- a *second-order schematic atom* P**`(`**$T_1, \ldots, T_n$**`)`** where P is a (second-order) predicate variable symbol and $T_1, \ldots, T_n$ are *schematic terms* in their explicit argument position form. Predicate variable symbols are represented exactly like Prolog variables, starting with an *upper-case* letter followed by any number of letters, digits and underscore symbols;
- a *vector of atoms* of the form **`{goals}`** or **`{goals*}`** – only components of the schematic body can be vectors of goals. In order to make a distinction between different vectors of atoms within the same schematic clause, numbers can be added on to the **`goals`** and **`goals*`** symbols.

Additionally, a component G of the *schematic body* can be

- a *truth value* of the form **`true`**, or
- a *schematic arithmetic expression* of the form x**`#1 is 'F'(&`**n**`,`**y**`,&`**m**`)#2`**, or

- a *schematic system test* `'Tn'(`$T_1, \ldots, T_n$`)` where n is an integer to differentiate distinct schematic tests within a schema and may not be used and $T_1,\ldots,T_n$ are *schematic terms* in their explicit argument position form.

A *schematic term* T is recursively defined as either

- a Prolog atom;
- a Prolog variable, including anonymous variables denoted by the underscore symbol;
- a *first-order schematic term* f`(`$T_1, \ldots, T_n$`)` where p is a Prolog functor symbol, $T_1,\ldots,T_n$ are also schematic terms;
- a *schematic constant* of the form `ct(`n`)`, where n is a natural number;
- a *second-order schematic term* F`(`$T_1, \ldots, T_n$`)` where F is a (second-order) functor variable symbol and $T_1, \ldots, T_n$ are also schematic terms. Functor variable symbols are represented exactly like Prolog variables;
- a *vector of terms* of the form `&`n, where n is a natural number.

Some examples of schematic terms are `[]`, `[X|Xs]`, `ct(1)` and `F(Y,&1,Ys)`. The *explicit argument position form* of a schematic term T, which is not a vector of terms, is $T\#V$, where V denotes its position within an atom, and is either a Prolog variable, an integer or an underscore; a vector of terms is already in its explicit argument position form. Some examples of schematic atoms with schematic terms in their explicit form are `sum(ct(1)#1,ct(2)#2)`, `p(&1,[]#N,&2,B(X,&3,Xs)#M,&4)` and `q(&1)`. An example of a complete enhanced schema is

```
P(&11,ct(1)#N,&12):-
      {goals11}
P(&21,F(&22,Xs,&23)#N,&24):-
      {goals21}
      P(&27,Xs#N,&28),
      {goals22}
```

A.2 Explicit Argument Position Format of Logic Programs

Since our schema language employ explicit argument positions in schematic atoms, the programs to be considered should also be in this format. A program **P** can be related to its explicit argument position form $\mathbf{P}^e$ in a simple way, formally depicted by the predicate *exp* below:

$$exp(\langle\,\rangle,\langle\,\rangle) \quad \vee$$

$$exp(\mathbf{P},\mathbf{P}^e) \Leftrightarrow \mathbf{P} = \langle C_i, \ldots, C_n\rangle \wedge \mathbf{P}^e = \langle C^e_i, \ldots, C^e_n\rangle \wedge exp^C(C_i,C^e_i) \wedge exp(\langle C_{i+1},\ldots, C_n\rangle,\langle C^e_{i+1},\ldots, C^e_n\rangle)$$

Each clause C_i is mapped onto its explicit argument position form C^e_i by the exp^C relationship, defined as

$$exp^C((H \,\texttt{:-}\, G_1,\ldots,G_n),(H^e \,\texttt{:-}\, G^e_1,\ldots,G^e_n)) \;\Leftrightarrow\; exp^G(H,H^e) \wedge exp^B(\langle G_1, \ldots,G_n\rangle,\langle G^e_1, \ldots,G^e_n\rangle)$$

that is, the head goal H is mapped onto its explicit argument position form H^e by means of the exp^G relationship, explained below, and the body comprised of a non-empty sequence of subgoals is mapped to its explicit argument position form by means of exp^B, also explained below. Predicate exp^G, relating a subgoal (first argument) to its explicit argument position form (second argument), is defined as

$$exp^G(p(t_1,\ldots,t_n),p(t_1\#1,\ldots,t_n\#n)) \;\vee exp^G(\texttt{true},\texttt{true})$$

the first element of the disjunction depicts the explicit argument position version of atoms: the predicate symbol p remains unchanged and each of its terms t_i is added a "#" followed by its argument position i. The second element states that constants **true** are mapped onto themselves, *i.e.* the explicit argument position form of **true** is **true** itself. Finally, the relationship exp^B maps the body of a program to its explicit argument form, by relating each subgoal to its corresponding explicit argument position form:

$exp^B(\langle\,\rangle, \langle\,\rangle) \quad \vee$

$exp^B(\langle G_i, \ldots, G_n\rangle, \langle G^e_i, \ldots, G^e_n\rangle) \Leftrightarrow exp^G(G_i, G^e_i) \wedge exp^B(\langle G_{i+1}, \ldots, G_n\rangle, \langle G^e_{i+1}, \ldots, G^e_n\rangle)$

The *exp* relationship can be employed to obtain the explicit argument position format of a conventional logic program (**P** is supplied and $\mathbf{P}^e$ is obtained by *exp*), to obtain the conventional form of a program in its explicit argument position form ($\mathbf{P}^e$ is supplied and **P** is obtained by *exp*) or to test if a conventional logic program and a program in explicit argument position form are related (both $\mathbf{P}^e$ and **P** are supplied).

A.3 Semantics: Schema Substitutions

Schemata are abstract descriptions of actual Prolog programs. The semantics of a schema can be given by listing each program it represents. This process can be infinite: simple variations of the same program with different variable symbols are all part of it. A more practical and feasible way to address the issue of meaning of a schema is to define a relation between it and candidate programs.

Given a schema and a program in the explicit argument position format, such that the schema is a generic representation of the program or, conversely, the program is an instance of the schema, then a schema substitution Θ, relating each schema symbol to a program construct can be obtained: when the schema symbols are replaced by their related program constructs the program is obtained. A schema substitution Θ is a possibly empty set of pairs $s_{\mathbf{S}}/s_{\mathbf{P}}$ where $s_{\mathbf{S}}$ is a schema symbol and $s_{\mathbf{P}}$ a program construction: this relationship is subject to constraints, depending on the kind of schema symbol, *i.e.* if it is a predicate variable, an argument position, and so on, and on the kind of program construction, *i.e.* if it is a predicate name, a constant, and so on. These relationships are stated below together with the formulation of a method to obtain a schema substitution unifying a schema and a program, if it exists.

Let there be a schema **S** consisting of the sequence of clauses $\langle C^{\mathbf{S}}_1, \ldots, C^{\mathbf{S}}_n\rangle$ and a program $\mathbf{P}^e$ in the explicit argument position format, consisting of clauses $\langle C^{\mathbf{P}}_1, \ldots, C^{\mathbf{P}}_n\rangle$. We say that **S** and $\mathbf{P}^e$ are unifiable if there is a schema substitution Θ associating each schema construct with parts of the program, such that if the parts of the program associated with the schema constructs are replaced in the schema then the program will be obtained. We define a relation *unify* between a schema and a program which holds if they are unifiable – this relation also obtains the schema substitution Θ, and it is formalised as

$$unify(\mathbf{S}, \mathbf{P}^e, \Theta) \Leftrightarrow unify(\mathbf{S}, \mathbf{P}^e, \varnothing, \Theta)$$

where the empty set stands for an auxiliary temporary schema substitution employed as the final substitution Θ is being prepared. The auxiliary *unify* relation is defined as follows

$unify(\langle\,\rangle, \langle\,\rangle, \Theta, \Theta) \quad \vee$

$unify(\mathbf{S}, \mathbf{P}^e, \Theta', \Theta) \Leftrightarrow \mathbf{S} = \langle C^{\mathbf{S}}_1, \ldots, C^{\mathbf{S}}_n\rangle \wedge \mathbf{P}^e = \langle C^{\mathbf{P}}_1, \ldots, C^{\mathbf{P}}_n\rangle \wedge unify^C(C^{\mathbf{S}}_1, C^{\mathbf{P}}_1, \Theta', \Theta'') \wedge$
$unify(\langle C^{\mathbf{S}}_2, \ldots, C^{\mathbf{S}}_n\rangle, \langle C^{\mathbf{P}}_2, \ldots, C^{\mathbf{P}}_n\rangle, \Theta'', \Theta)$

The first element of the disjunction describes the case when the schema and the program are empty sequences of clauses: in such circumstances the final schema substitution is equal to the auxiliary substitution. The second element of the disjunction describes the case when the schema **S** and the program **P** are non-empty sequences of clauses with the same number of clauses: the clause $C^{\mathbf{S}}_1$ of the schema must then satisfy the relation $unify^C$ together with its corresponding clause $C^{\mathbf{P}}_1$ of the program. The relation $unify^C$ employs the auxiliary schema substitution Θ' in order to obtain a new auxiliary substitution Θ'', used recursively to match the rest of the clauses.

The relationship $unify^C$ between two clauses is defined as follows:

$$unify^C(C^s, C^p, \Theta', \Theta) \Leftrightarrow C^s = H^s \texttt{ :- } G^s_1, \ldots, G^s_n \wedge C^p = H^p \texttt{ :- } G^p_1, \ldots, G^p_m \wedge unify^G(H^s, H^p, \Theta', \Theta'') \wedge unify^B(\langle G^s_1, \ldots, G^s_n\rangle, \langle G^p_1, \ldots, G^p_m\rangle, H^p, \Theta'', \Theta)$$

that is, two clauses C^s and C^p satisfy the relation $unify^C$, given an initial schema substitution Θ', if their head goals H^s and H^p satisfy the relation $unify^G$ and their bodies satisfy the relation $unify^B$. We shall assume that a fact in program **P** will be represented as a clause with a **`true`** body.

The body of a clause of the schema is related via $unify^B$ to the body of the corresponding clause of the program: the schematic subgoals and the actual concrete constructions should match appropriately. This relation is defined in a recursive fashion, in terms of the unification of their corresponding subgoals:

$$unify^B(\langle\,\rangle, \langle\,\rangle, H^p, \Theta, \Theta) \quad \vee$$

$$unify^B(B^s, B^p, H^p, \Theta', \Theta) \Leftrightarrow B^s = \langle G^s_i, \ldots, G^s_n\rangle \wedge G^s_i \neq \texttt{\{goals}n\texttt{\}} \wedge G^s_i \neq \texttt{\{goals*}n\texttt{\}} \; B^p = \langle G^p_j, \ldots, G^p_m\rangle \wedge unify^G(G^s_i, G^p_j, \Theta', \Theta'') \wedge unify^B(\langle G^s_{i+1}, \ldots, G^s_n\rangle, \langle G^p_{j+1}, \ldots, G^p_m\rangle, H^p, \Theta'', \Theta) \quad \vee$$

$$unify^B(B^s, B^p, H^p, \Theta', \Theta) \Leftrightarrow B^s = \langle \texttt{\{goals}n\texttt{\}}, G^s_2, \ldots, G^s_n\rangle \wedge B^p = B^p_1 \cdot B^p_2 \wedge non\text{-}recursive(B^p_1, H^p) \wedge unify^\Theta(\texttt{\{goals}n\texttt{\}}, B^p_1, \Theta', \Theta'') \wedge unify^B(\langle G^s_2, \ldots, G^s_n\rangle, B^p_2, H^p, \Theta'', \Theta) \quad \vee$$

$$unify^B(B^s, B^p, H^p, \Theta', \Theta) \Leftrightarrow B^s = \langle \texttt{\{goals*}n\texttt{\}}, G^s_2, \ldots, G^s_n\rangle \wedge B^p = B^p_1 \cdot B^p_2 \wedge unify^\Theta(\texttt{\{goals*}n\texttt{\}}, B^p_1, \Theta', \Theta'') \wedge unify^B(\langle G^s_2, \ldots, G^s_n\rangle, B^p_2, H^p, \Theta'', \Theta)$$

The first element of the disjunction depicts the base case of the recursion, when empty sequences of subgoals are eventually obtained. The second element of the disjunction depicts the unification of a schematic atom G^s_i which is not a vector of atoms: it should unify with the corresponding subgoal G^p_j of the program. Vectors of atoms deserve a special treatment because they unify with subsequences of the program's subgoals – the third and fourth cases of the disjunction cater for this possibility. For both **`{goals`*n*`}`** and **`{goals*`*n*`}`**, the sequence B^p of subgoals of the program is split into two parts B^p_1 and B^p_2; for **`{goals`*n*`}`**, B^p_1 is tested for recursive subgoals in predicate *non-recursive*: recursive subgoals are *not* allowed in the unification of vectors of atoms **`{goals`*n*`}`**, and this explains the presence of the head goal H^p of the clause in the relation above. Vectors of atoms **`{goals*`*n*`}`** do not have any constraints as to what subgoals may be unified with them, hence the absence of the *non-recursive* test.

The $unify^\Theta$ relation checks if there already is a binding between the schema constructs and parts of the program:

$$unify^\Theta(T^s, T^p, \Theta, \Theta) \Leftrightarrow T^s/T \in \Theta \wedge T = T^p \quad \vee$$

$$unify^\Theta(T^s, T^p, \Theta', \Theta) \Leftrightarrow T^s/T \notin \Theta' \wedge (\Theta = \Theta' \cup \{T^s/T^p\})$$

the first element of the definition above states that if the schema construct T^s is already bound to a construct T in Θ, then T^s will only unify with T^p if T is equivalent to T^p; the second element of the definition states that if T^s is not bound yet, then it unifies with T^p and a new schema substitution Θ is obtained by inserting the pair T^s/T^p to the old substitution Θ'.

The unification of schematic atoms and program subgoals is performed by the $unify^G$ relation, addressing the different schematic constructs and relating them to actual parts of the program:

$unify^G(\mathbf{true},\mathbf{true},\Theta,\Theta) \quad \vee$

$unify^G(p\,\mathbf{(}T^S{}_1,\ldots,T^S{}_n\mathbf{)}\,,p\,\mathbf{(}T^P{}_1,\ldots,T^P{}_m\mathbf{)}\,,\Theta',\Theta) \Leftrightarrow unify^T(\langle T^S{}_1,\ldots,T^S{}_n\rangle,\langle T^P{}_1,\ldots,T^P{}_m\rangle,\Theta',\Theta) \quad \vee$

$$\begin{aligned} unify^G(\mathbf{'T}n\mathbf{'}\,\mathbf{(}T^S{}_1,\ldots,T^S{}_n\mathbf{)}\,,p\,\mathbf{(}T^P{}_1,\ldots,T^P{}_m\mathbf{)}\,,\Theta',\Theta) \Leftrightarrow\ & system\text{-}test(p^m) \wedge \\ & unify^\Theta(\mathbf{'T}n\mathbf{'},p,\Theta',\Theta'') \wedge \\ & unify^T(\langle T^S{}_1,\ldots,T^S{}_n\rangle,\langle T^P{}_1,\ldots,T^P{}_m\rangle,\Theta'',\Theta) \quad \vee \end{aligned}$$

$$\begin{aligned} unify^G(x\mathbf{\#1\ is\ 'F'\,(\&}n\mathbf{,}\,y\mathbf{,\,\&}m\mathbf{)\,\#2},z\mathbf{\#1\ is}\,f(g(t_1,\ldots,t_k), w, h(s_1,\ldots,s_l))\mathbf{\#2},\Theta',\Theta) \Leftrightarrow & \\ & unify^\Theta(x, z, \Theta',\Theta'') \wedge \\ & unify^\Theta(\mathbf{\&}n, g(t_1,\ldots,t_k), \Theta'',\Theta^3) \wedge \\ & unify^\Theta(y, w, \Theta^3,\Theta^4) \wedge \\ & unify^\Theta(\mathbf{\&}m, h(s_1,\ldots,s_l), \Theta^4,\Theta) \quad \vee \end{aligned}$$

$$\begin{aligned} unify^G(P\,\mathbf{(}T^S{}_1,\ldots,T^S{}_n\mathbf{)}\,,p\,\mathbf{(}T^P{}_1,\ldots,T^P{}_m\mathbf{)}\,,\Theta',\Theta) \Leftrightarrow\ & unify^\Theta(P,p,\Theta',\Theta'') \wedge \\ & unify^T(\langle T^S{}_1,\ldots,T^S{}_n\rangle,\langle T^P{}_1,\ldots,T^P{}_m\rangle,\Theta'',\Theta) \end{aligned}$$

The first element of the disjunction states that a truth value only unifies with another similar truth value; the second element states that a first-order schematic atom unifies with a subgoal if they have the same predicate symbol p and their terms satisfy the relation $unify^T$ described below; the third element of the disjunction states that schema constructs employing **'T***n***'** predicate symbols unify with system test predicates p^m (*ground*/1, *var*/1, *number*/1, **>**, **<**, and so on) if their terms satisfy the relation $unify^T$; the fourth element depicts the unification between the schematic representation of arithmetic expressions and a concrete program construction employing the **is** Prolog predicate; the fifth element of the disjunction states that a second-order schematic atom unifies with a subgoal if the second-order predicate variable P of the schematic construct and the predicate symbol p of the subgoal satisfy the $unify^\Theta$ relation and their terms satisfy the relation $unify^T$.

The $unify^T$ relation recursively applies the $unify^{terms}$ predicate to pairs of corresponding schematic terms and concrete program constructions:

$unify^T(\langle\,\rangle, \langle\,\rangle,\Theta,\Theta) \quad \vee$

$$\begin{aligned} unify^T(\langle \mathbf{\&}n, T^S{}_i,\ldots,T^S{}_n\rangle,\mathbf{T}, \Theta', \Theta) \Leftrightarrow\ & \mathbf{T} = \mathbf{T}_1 \cdot \mathbf{T}_2 \ \wedge\ unify^\Theta(\mathbf{\&}n,\mathbf{T}_1,\Theta',\Theta'') \wedge \\ & unify^T(\langle T^S{}_i,\ldots,T^S{}_n\rangle,\mathbf{T}_2, \Theta'', \Theta) \quad \vee \end{aligned}$$

$$\begin{aligned} unify^T(\langle T^S,T^S{}_i,\ldots,T^S{}_n\rangle,\langle T^P,T^P{}_j,\ldots,T^P{}_m\rangle, \Theta', \Theta) \Leftrightarrow\ & unify^{terms}(T^S,T^P,\Theta',\Theta'') \wedge \\ & unify^T(\langle T^S{}_i,\ldots,T^S{}_n\rangle,\langle T^P{}_j,\ldots,T^P{}_m\rangle, \Theta'', \Theta) \end{aligned}$$

The vectors of terms **&***n* deserve special care, being unified with a subsequence $\mathbf{T}_1$ of the sequence $\mathbf{T}$ of program terms, as depicted in the second element of the disjunction above. The pairs of schematic and concrete terms are related by $unify^{terms}$ as follows:

$unify^{terms}(T^S\mathbf{\#}t,T^P\mathbf{\#}n, \Theta', \Theta) \Leftrightarrow unify^\Theta(t,n,\Theta',\Theta'') \ \wedge\ unify^{terms}(T^S,T^P, \Theta'', \Theta) \quad \vee$

$unify^{terms}(c, c, \Theta, \Theta) \quad \vee$

$unify^{terms}(x,T, \Theta', \Theta) \Leftrightarrow unify^\Theta(x,T,\Theta', \Theta) \quad \vee$

$unify^{terms}(\mathbf{ct\,(}n\mathbf{)}\,,c, \Theta', \Theta) \Leftrightarrow unify^\Theta(\mathbf{ct\,(}n\mathbf{)}\,,c,\Theta', \Theta) \quad \vee$

$unify^{terms}(f\,\mathbf{(}S^S{}_1,\ldots,S^S{}_r\mathbf{)}\,,f\,\mathbf{(}S^P{}_1,\ldots,S^P{}_v\mathbf{)}\,,\Theta',\Theta) \Leftrightarrow unify^T(\langle S^S{}_1,\ldots,S^S{}_r\rangle,\langle S^P{}_1,\ldots,S^P{}_v\rangle,\Theta', \Theta) \quad \vee$

$$\begin{aligned} unify^{terms}(F\,\mathbf{(}S^S{}_1,\ldots,S^S{}_r\mathbf{)}\,,f\,\mathbf{(}S^P{}_1,\ldots,S^P{}_v\mathbf{)}\,,\Theta',\Theta) \Leftrightarrow\ & unify^\Theta(F,f,\Theta',\Theta'') \wedge \\ & unify^T(\langle S^S{}_1,\ldots,S^S{}_r\rangle,\langle S^P{}_1,\ldots,S^P{}_v\rangle,\Theta',\Theta) \end{aligned}$$

The symbol c stands for a generic Prolog atom, x and y are generic Prolog variables and n is an integer.

Appendix B: Selected Schema-Based Program Transformations

In this section we list some selected schema-based program transformations offered by our system. We have only addressed those procedures with two clauses, for the sake of simplicity. The transformations are given their names according to their targeted constructs, *i.e.* if they are predicate- or conjunction-altering. In the former case, the transformation start with a "**p**"; in the latter, it starts with a "**c**". The number as the first argument is the heuristic value assigned to the transformation. The empty set as a conjunction informs our system that the alterations prescribed by that transformation are confined to the procedure and no external adaptations (*e.g.* changing the order of parameters, inserting new arguments, and so on) have to be performed.

The first two transformations **p01** and **p02** change the order of a test predicate. Since our schema language does not support the representation of permutations, a transformation is devised for each focused clause. Transformation **p01** focuses on the first clause:

```
⟨100,p01, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-               ›⟩
                  {goals*11},                    {goals*11},
                  P(&12,X#N,&13),                P(&12,X#N,&13),
                  {goals*12},                    {goals*12},
                  Q(&14),                        'T'(&15,X#_,&16),
                  {goals*13},                    Q(&14),
                  'T'(&15,X#_,&16),              {goals*13},
                  {goals*14}                     {goals*14}
               S(&21):-                       S(&21):-
                  {goals*21}                     {goals*21}
```

Transformation **p02** focuses on the second clause, performing similar changes:

```
⟨100,p02, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-               ›⟩
                  {goals*11}                     {goals*11}
               S(&21):-                       S(&21):-
                  {goals*21},                    {goals*21},
                  P(&22,X#N,&23),                P(&22,X#N,&23),
                  {goals*22},                    {goals*22},
                  Q(&24),                        'T'(&25,X#_,&26),
                  {goals*23},                    Q(&24),
                  'T'(&25,X#_,&26),              {goals*23},
                  {goals*24}                     {goals*24}
```

Transformations **p11** and **p12** below change the ordering of an explicit instantiation, transferring it to a point before a subgoal. This simple alteration may save much computing time by constraining the execution of procedure **P**. Transformation **p11** focuses on the first clause:

```
⟨100,p11, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-               ›⟩
                  {goals*11},                    {goals*11},
                  P(&12,X#_,&13),                X#1 = G(&14)#2,
                  {goals*12},                    P(&12,X#_,&13),
                  X#1 = G(&14)#2,                {goals*12},
                  {goals*13}                     {goals*13}
               S(&21):-                       S(&21):-
                  {goals*21}                     {goals*21}
```

Transformation **p12** focuses on the second clause:

```
⟨100,p12, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-               ›⟩
                  {goals*11}                     {goals*11}
               S(&21):-                       S(&21):-
                  {goals*21},                    {goals*21},
                  P(&22,X#_,&23),                X#1 = G(&24)#2,
                  {goals*22},                    P(&22,X#_,&23),
                  X#1 = G(&24)#2,                {goals*22},
                  {goals*23}                     {goals*23}
```

Transformations **p21** and **p22** below remove spurious calls to the *append/3* predicate, if the first argument is a singleton list **[X]**. Transformation **p21** prescribes the appropriate alterations when the focus of the analysis is the first clause:

```
⟨100,p21, ∅, ‹ S(&11):-                    ›, ∅, ‹ S(&11):-          › ⟩
                   {goals*11},                        {goals*11},
                   append([X]#1,Y#2,Z#3),             Z#1 = [X|Y]#2,
                   {goals*12}                         {goals*12}
               S(&21):-                           S(&21):-
                   {goals*21}                         {goals*21}
```

Transformation **p22** addresses the second clause:

```
⟨100,p22, ∅, ‹ S(&11):-                    ›, ∅, ‹ S(&11):-          › ⟩
                   {goals*11}                         {goals*11}
               S(&21):-                           S(&21):-
                   {goals*21},                        {goals*21},
                   append([X]#1,Y#2,Z#3),             Z#1 = [X|Y]#2,
                   {goals*22}                         {goals*22}
```

Transformations **p31** and **p32** remove explicit instantiations by unifying them and propagating the unification throughout the clause. When the focus of the analysis is the first clause, then transformation **p31** should be used:

```
⟨100,p31, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-                 › ⟩
                   {goals*11},                     unify(X,G(&12)),
                   X#1 = G(&12)#2,                 {goals*11},
                   {goals*12}                      {goals*12}),
               S(&21):-                        S(&21):-
                   {goals*21}                      {goals*21}
```

When the focus of the analysis is the second clause, then transformation **p32** should be used:

```
⟨100,p32, ∅, ‹ S(&11):-                 ›, ∅, ‹ S(&11):-                 › ⟩
                   {goals*11}                      {goals*11}
               S(&21):-                        S(&21):-
                   {goals*21},                     unify(X,G(&22)),
                   X#1 = G(&22)#2,                 {goals*21},
                   {goals*22}                      {goals*22}
```

Transformations **p41** and **p42** below insert an accumulator pair in a procedure performing a non-tail recursive arithmetic computation, thus making the procedure iterative. A generic description of a singly-recursive data-structure has been employed, thus conveying more expressiveness to the transformation. Different transformations had to be provided, though, to address alternative relative orderings of argument positions. Transformation **p41** describes procedures in which the data structure is to the left of the argument the arithmetic computation is assigned to:

```
⟨20,p41, ‹S(&1,L#N,&2,R#M,&3)›, ‹ S(&11,ct(1)#N,&12,ct(2)#M,&13):-
                                       {goals11}
                                   S(&21,G(X,Xs)#N,&22,Res#M,&23):-
                                       {goals21},
                                       S(&24,Xs#N,&25,XsRes#M,&26),
                                       {goals22},
                                       Res#1 is 'F'(&27,XsRes,&28)#2,
                                       {goals23}                          ›,

‹S_p4(&1,L#N,&2,ct(2)#_,R#_,&3)›, ‹ S_p4(&11,ct(1)#N,&12,FAc#_,FAc#_,&13):-
                                        {goals11}
                                    S_p4(&21,G(X,Xs)#N,&22,TAc#_,Ac#_,&23):-
                                        {goals21},
                                        NAc#1 is 'F'(&27,TAc,&28)#2,
                                        S_p4(&24,Xs#N,&25,NAc#_,Ac#_,&26),
                                        {goals22},
                                        {goals23}                            › ⟩
```

Transformation **p42** describes procedures in which the data structure is to the right of the argument the arithmetic computation is assigned to:

```
⟨ 20,p42, ‹S(&1,R#M,&2,L#N,&3)›, ‹
S(&11,ct(1)#M,&12,ct(2)#N,&13):-
    {goals11}
S(&21,Res#M,&22,G(X,Xs)#N,&23):-
    {goals21},
    S(&24,XsRes#M,&25,Xs#N,&26),
    {goals22},
    Res#1 is 'F'(&27,XsRes,&28)#2,
    {goals23}
›,
‹S_p4(&1,ct(1)#_,R#_,&2,L#_,&3)›, ‹
S_p4(&11,FAc#_,FAc#_,&12,ct(2)#_,&13):-
    {goals11}
S_p4(&21,TAc#_,Ac#_,&22,G(X,Xs)#_,&23):-
    {goals21},
    NAc#1 is 'F'(&27,TAc,&28)#2,
    S_p4(&24,NAc#_,Ac#_,&25,Xs#_,&26),
    {goals22},
    {goals23}
› ⟩
```

Transformations **p43** and **p44** are similar to the preceding pair of transformations, but in this case computations employing *append*/3 are addressed. Transformation **p43** describes procedures in which the data structure is to the left of the argument the *append* computation is assigned to:

```
⟨ 20,p43, ‹S(&1,L#N,&2,R#M,&3)›, ‹
S(&11,ct(1)#N,&12,ct(2)#M,&13):-
    {goals11}
S(&21,G(X,Xs)#N,&22,Res#M,&23):-
    {goals21},
    S(&24,Xs#N,&25,XsRes#M,&26),
    {goals22},
    append(XsRes#1,V#2,Res#3),
    {goals23}
›,
‹S_p4(&1,L#N,&2,ct(2)#_,R#_,&3)›, ‹
S_p4(&11,ct(1)#N,&12,FAc#_,FAc#_,&13):-
    {goals11}
S_p4(&21,G(X,Xs)#N,&22,TAc#_,Ac#_,&23):-
    {goals21},
    append(V#1,TAc#2,NAc#3),
    S_p4(&24,Xs#N,&25,NAc#_,Ac#_,&26),
    {goals22},
    {goals23}
› ⟩
```

Transformation **p44** describes procedures in which the data structure is to the right of the argument the *append* computation is assigned to:

```
⟨ 20,p44, ‹S(&1,R#M,&2,L#N,&3)›, ‹
S(&11,ct(1)#M,&12,ct(2)#N,&13):-
    {goals11}
S(&21,Res#M,&22,G(X,Xs)#N,&23):-
    {goals21},
    S(&24,XsRes#M,&25,Xs#N,&26),
    {goals22},
    append(XsRes#1,V#2,Res#3),
    {goals23}
›,
‹S_p4(&1,ct(1)#_,R#_,&2,L#_,&3)›, ‹
S_p4(&11,FAc#_,FAc#_,&12,ct(2)#_,&13):-
    {goals11}
S_p4(&21,TAc#_,Ac#_,&22,G(X,Xs)#_,&23):-
    {goals21},
    append(V#1,TAc#2,NAc#3),
    S_p4(&24,NAc#_,Ac#_,&25,Xs#_,&26),
    {goals22},
    {goals23}
› ⟩
```

Below we have the first transformation **c1** aimed at conjunctions: it merges two procedures manipulating the same singly-recursive data structure. The generic representation for the data structure ensures that **c1** will be applied to any singly-recursive data structure with two clauses:

```
〈30,c1, ‹S1(&1,L#N,&2),S2(&3,L#M,&4)›,

‹ S1(&111,ct(1)#N,&112):-
      {goals111}
  S1(&121,G(X,Xs)#N,&122):-
      {goals121},
      S1(&123,Xs#N,&124),
      {goals122}
,
  S2(&211,ct(1)#M,&212):-
      {goals211}
  S2(&221,G(Y,Ys)#M,&222):-
      {goals221},
      S2(&223,Ys#M,&224),
      {goals222}
›,

‹S1_S2_c1(&1,&3,L#_,&2,&4)›,
‹ S1_S2_c1(&111,&211,ct(1)#_,&112,&212):-
      {goals111},
      {goals211}
  S1_S2_c1(&121,&221,G(X,Xs)#_,&122,&222):-
      {goals121},
      {goals221},
      S1_S2_c1(&123,&223,Xs#_,&124,&224),
      {goals122},
      {goals222}
›〉
```

Transformation **c2** merges two data-structure manipulating procedures, inserting an extra clause in order to address the different possibilities of recursions:

```
〈10,c2, ‹S1(&1,V1#N1,&2,V2#N2,&3,V3#N3,&4),S2(&5,V3#M,&6)›,

‹ S1(&111,ct(1)#N1,&112,W1#N2,&113,W1#N3,&114):-
      {goals111}
  S1(&121,G(X,Xs)#N1,&122,W2#N2,&123,G(X,Ys)#N3,&124):-
      {goals121},
      S1(&125,Xs#N1,&126,W2#N2,&127,Ys#N3,&128),
      {goals122}
,
  S2(&211,ct(1)#M,&212):-
      {goals211}
  S2(&221,G(X,Xs)#M,&222):-
      {goals221},
      S2(&223,Xs#M,&224),
      {goals222}
›,

‹S1_S2_c2(&1,V1#_,&2,V2#_,&3,V3#_,&4,&5,&6)›,

‹ S1_S2_c2(&111,ct(1)#_,&112,ct(1)#_,&113,ct(1)#_,&114,&211,&212):-
      {goals111},
      {goals211}
  S1_S2_c2(&111,ct(1)#_,&112,G(X,Xs)#_,&113,G(X,Xs)#_,&114,&221,&222):-
      {goals111},
      {goals221},
      S1_S2_c2(&111,ct(1)#_,&112,Xs#_,&113,Xs#_,&114,&223,&224),
      {goals222}
  S1_S2_c2(&121,G(X,Xs)#_,&122,W2#_,&123,G(X,Ys)#_,&124,&221,&222):-
      {goals121},
      {goals221},
      S1_S2_c2(&125,Xs#_,&126,W2#_,&127,Ys#_,&128,&223,&224),
      {goals122},
      {goals222}
›〉
```

An interesting example of the application of this transformation is given below. The original program is

```
sum([],0).
sum([X|Xs],S):-
      sum(Xs,SXs),
      S is SXs + X.

append([],L,L).
append([X|Xs],L,[X|Ys]):-
      append(Xs,L,Ys).

p(L1,L2,L3,S):-
      append(L1,L2,L3),
      sum(L3,S).
```

The resulting program, after the appropriate application of the transformation, is:

```
sum([],0).
sum([X|Xs],S):-
      sum(Xs,SXs),
      S is SXs + X.

append([],L,L).
append([X|Xs],L,[X|Ys]):-
      append(Xs,L,Ys).

p(L1,L2,L3,S):-
      append_sum_c2(L1,L2,L3,S).

append_sum_c2([],[],[],0).
append_sum_c2([],[X|Xs],[X|Xs],S):-
      append_sum_c2([],Xs,Xs,SXs),
      S is SXs + X.
append_sum_c2([X|Xs],L,[X|Ys],S):-
      append_sum_c2(Xs,L,Ys,SXs),
      S is SXs + X.
```