

Schema-Based Transformations of Logic Programs

Norbert E. Fuchs, Markus P. J. Fromherz

Institut für Informatik
Universität Zürich
Switzerland
{fuchs, fromherz}@ifi.unizh.ch

Transformation schemata are predefined abstract transformations of logic programs: input program schemata are transformed into output program schemata. Each transformation schema represents one transformation strategy, for example a particular sequence of applications of the unfold/fold rules, or the introduction of an accumulator data structure. The transformation of logic programs with the help of transformation schemata proceeds in three steps: *abstraction* of the programs to program schemata, *selection* of a transformation schema with these schemata as input and a suitable schema as output, and *specialization* of the output schema to the transformed program. Once the transformation schemata are available, user intervention is required only during the selection step. For standard transformation situations one can even envisage eliminating user interaction altogether by heuristics.

1 Introduction

Logic programming allows to write software specifications and the specified programs in the same language. But even when written in the same language, specifications and programs must not be confused. Specifications should describe the functionality of programs in a declarative, clear way, which usually means that they are not efficiently executable, while the programs themselves should be efficiently executable. Logic programming promises to bridge the apparent conflict by providing means to transform specifications into programs. For this reason transformations of logic programs into equivalent, but more efficient forms have a rather long tradition.

Often these transformations are based on unfold/fold rules which were introduced by [Burstall & Darlington 77] in the context of functional programs. [Tamaki & Sato 84] defined transformations of definite logic programs based on unfolding and folding. Their transformations generate equivalent programs in that the least Herbrand model and the computed answers are preserved. [Gardner & Shepherdson 89] introduced slightly modified transformations for normal logic programs which preserve procedural semantics for SLDNF resolution and declarative semantics based on Clark's completion.

Though the unfold/fold rules preserve the semantics they cannot be used blindly. Their application requires user intervention and thus prevents transformations to be automated. Different strategies have been proposed to overcome practical problems of individual rules and to semi-automate transformations [Lakhotia & Sterling 88, Nielson & Nielson 90, Pereira & Shieber 87, Proietti & Pettorossi 90]. However, these strategies do not lead easily to strategies for transformations consisting of several applications of unfold/fold rules.

We suggest a radically different approach that is based on transformation schemata. Transformation schemata are predefined abstract transformations which transform schemata of logic programs into other schemata. Each transformation schema represents a transformation strategy, e.g. a particular sequence of applications of unfold/fold or other transformation rules. A set of transformation schemata constitutes an extensible transformation system. The transformation of individual programs is reduced to the search for an appropriate transformation schema. This approach allows considerable reduction and simplification of user interaction since users don't have to deal with the intricacies of the unfold/fold rules and can concentrate on the form of the transformed programs alone. Standard transformations promise to be completely automated by replacing user interaction by heuristics.

This paper has the following structure. In section 2 we define the three unfold/fold rules *definition*, *unfolding* and *folding* which we use in section 3 for an example transformation. Section 4 sketches practical problems with these unfold/fold rules and solutions suggested for some of the problems. In section 5 we introduce schemata for logic programs and briefly describe the set of schemata developed by [Gegg-Harrison 89]. Section 6 defines transformation schemata as transformations of program schemata, and shows how programs can be transformed with the help of transformation schemata in the three steps *abstraction*, *selection*, and *specialization*. In section 7 we describe the implementation of an experimental transformation system based on transformation schemata. Section 8 contains examples of transformation schemata and of concrete program transformations. Finally, in section 9 we summarize the main results and indicate directions for further research.

2 Unfold/Fold Rules

Three of the unfold/fold transformation rules are especially important and are often used in combination: the definition of new predicates in terms of given predicates, unfolding new predicates with respect to the clauses of the given predicates, and folding of literals generated by unfolding. We will briefly define these three rules following [Gallagher 90] and [Gardner & Shepherdson 89]. Other transformation rules, e.g. goal replacement, more specific clauses, and using laws of the predicates, will not be discussed.

New Definition

Let P be a normal program and D a set of clauses. Define S as the new set of clauses $p(\dots) \leftarrow Q_1, \dots, p(\dots) \leftarrow Q_n$ where p is a predicate symbol not occurring in P , D , or Q_1, \dots, Q_n . By letting $P' = P \cup S$ and $D' = D \cup S$ we transform P and D into P' and D' .

Unfolding

Let P be a normal program and C a clause in P of the form $A \leftarrow Q_1, B, Q_2$ where A and B are atoms and Q_1 and Q_2 conjunctions of literals. Let $H_1 \leftarrow R_1, \dots, H_n \leftarrow R_n$ be the clauses in P whose heads H_i unify with B with the mgu's $\theta_1, \dots, \theta_n$. Unfolding C on B generates the clauses $(A \leftarrow Q_1, R_1, Q_2)\theta_1, \dots, (A \leftarrow Q_1, R_n, Q_2)\theta_n$. Replacing C by these clauses transforms the program P into the program P' .

Folding

Let P be a normal program and D a set of clauses introduced by the new definition rule. Let C be a clause in P of the form $A \leftarrow Q\theta, R$ where Q and R are conjunctions of literals, and θ a substitution. Let C_1 be a clause $H \leftarrow Q$ in D which is not a variant of C . Folding C using C_1 generates the clause C_2 of the form $A \leftarrow H\theta, R$ provided that unfolding C_2 on $H\theta$ with respect to D gives C , C_1 is the only clause in D whose head unifies with $Q\theta$, and θ maps variables which appear in Q , but not in H into distinct variables which do not occur in C_2 . Replacing C by C_2 transforms P into P' .

3 An Example Transformation

As a basis for the subsequent discussion we present a standard example of the unfold/fold transformations. The predicate *average/2* calculates the average value of the elements of a list.

```
average([], 0).
average(List, Average) :-
    sum(List, Sum),
    length(List, Length),
    Average is Sum/Length.
```

Where the predicates *sum/2* and *length/2* are defined as

```
sum([], 0).
sum([First | Rest], Sum) :-
    sum(Rest, RestSum),
    Sum is RestSum + First.
```

```
length([], 0).
length([First | Rest], Length) :-
    length(Rest, RestLength),
    Length is RestLength + 1.
```

As defined, *average/2* is inefficient because it traverses the list twice to calculate its sum and its length though one traversal would suffice. To calculate sum and length in one traversal we define a new predicate *sumlength/3* as a composition of *sum/2* and *length/2*.

```
sumlength(List, Sum, Length) :-                               % A
    sum(List, Sum),
    length(List, Length).
```

Unfolding *sumlength/3* on *sum/2* yields the two clauses

```
sumlength([], 0, Length) :-                                  % B1
    length([], Length).
sumlength(First | Rest, Sum, Length) :-                     % B2
    sum(Rest, RestSum),
    Sum is RestSum + First,
    length(First | Rest, Length).
```

and unfolding these two clauses on *length/2* gives

```
sumlength([], 0, 0).                                % C1
sumlength([First | Rest], Sum, Length) :-          % C2
    sum(Rest, RestSum),
    Sum is RestSum + First,
    length(Rest, RestLength),
    Length is RestLength + 1.
```

Now we fold the conjunction $sum(Rest, RestSum), length(Rest, RestLength)$ of the clause C2 using the definition A of *sumlength/3*. We get

```
sumlength([First | Rest], Sum, Length) :-          % D2
    sumlength(Rest, RestSum, RestLength),
    Sum is RestSum + First,
    Length is RestLength + 1.
```

The predicate *sumlength/3* is now defined by the clauses C1 and D2, and a new version of *average/2* can be derived by folding $sum(List, Sum), length(List, Length)$.

```
average([], 0).
average(List, Average) :-
    sumlength(List, Sum, Length),
    Average is Sum/Length.
```

The new version of *average/2* runs about 30% faster than the old one; in other cases, transformations by the unfold/fold rules can lead to a much larger gain in efficiency.

4 Practical Problems with the Unfold/Fold Rules

The preceding transformation example looks straightforward, but in fact involves several careful decisions that amount to a transformation strategy. Following a different strategy may have led us astray. Many practical problems prevent unfold/fold transformations from being performed routinely, or even automatically.

The definition of new predicates – also called *eureka* rule – is a creative act which requires our intuition and discretion. Introducing the predicate *sumlength/3* seems obvious, but how we got the idea in the first place remains unclear. Recently, attempts have been made to derive eureka definitions in a systematic way. [Proietti & Pettorossi 90] introduced two methods to find eureka predicates. They applied these methods successfully to some classes of logic programs. In the realm of functional programs, [Nielson & Nielson 90] showed that type information can be used to derive eureka definitions.

Unfolding is often possible in many different ways. In our example we unfolded *sumlength/3* on *sum/2* and the resulting clauses B1 and B2 on *length/2*.

Instead, we could have unfolded the clause B2 on *sum/2* to get

```

sumlength([First], Sum, Length) :-                % E2
    Sum is First,
    length([First], Length).
sumlength([First, Second | Rest], Sum, Length) :- % E3
    sum(Rest, RestSum),
    RestSum1 is RestSum + Second,
    Sum is RestSum1 + First,
    length([First, Second | Rest], Length).

```

The predicate *sumlength/3* is now defined by the clauses B1, E2, and E3.

Though the clauses E2 and E3 may be more efficient than B2, they are unnecessarily specific: E2 treats lists with one, and E3 lists with at least two elements. Further unfolding of B1, E2, and E3 yields additional clauses, most of which are quite useless since they deal with additional special cases. Unfolding increases the number of clauses and can even lead to a combinatorial explosion.

This means that human intervention is required to control unfolding, and that there is a great need for guidelines. For the case of specializing an interpreter by unfolding it with respect to an object program two strategies have been proposed. [Pereira & Shieber 87] suggested to divide predicates into evaluable and residual ones. [Lakhotia & Sterling 88] proposed to restrict the unfolding of the interpreter to its parsing component and to leave its execution component as residue. These strategies reduce and facilitate human intervention but do not eliminate it.

Folding also presents problems. The folding rule assumes that the literals to be folded are consecutive. In fact, in clause C2 the two literals – *sum(Rest, RestSum)* and *length(Rest, RestLength)* – are not consecutive. The independence of the computation rule of SLD resolution allows to rearrange the literals, but Prolog's left-to-right computation rule could lead to termination problems. If the literals to be folded are not consecutive it is also not clear where to put the literal resulting from the folding step. In clause D2 we carefully ordered the literals so that the predicate *sumlength/3* can be evaluated without generating an error message.

As we have seen, human assistance is required for all three transformation rules, thus preventing automatic transformations. One way to semi-automate transformations is to formulate human assistance as strategies for each transformation rule. In this paper we suggest a different approach based on transformation schemata. This approach allows to formulate strategies for complete sequences of transformation steps, thus reducing and simplifying user interaction. For standard transformation situations user interaction could be completely replaced by heuristics leading to automatic transformations.

5 Schemata for Logic Programs

Since the beginning of logic programming it has been recognized that many logic programs, e.g. list processing programs, are structured similarly, and can be understood as instances of program schemata.

Different sets of schemata have been proposed. [O'Keefe 90] defined a set of schemata for recursive programs, while [Deville & Burnay 90] suggested schemata as a basis for program construction derived from structural induction and generalization.

Schemata can also be used to abstract from programs which rely on the same programming technique, e.g. on accumulators or on difference lists. [Brna et al. 88] defined a large number of these techniques. [Robertson 91] showed how programming techniques can be used to teach the construction of logic programs. [Lakhotia 89] demonstrated that standard programming techniques can be incorporated into initial programs with the help of partial evaluation.

By far the most comprehensive set of schemata was introduced by [Gegg-Harrison 89] in the context of a tutoring system. Based on the notion of the most specific generalization, second-order schemata are produced. Applied to a large number of simple recursive list-processing Prolog programs, a hierarchy of Prolog schemata is created. At the top of this hierarchy are fourteen basic-level schemata which capture the majority of simple recursive list-processing Prolog programs.

To describe the schemata, Gegg-Harrison conceived a language which we will introduce by an example. The basic-level schema A classifies programs which recursively process all elements of a given list.

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

Arguments and literals in angle brackets < ... > are optional, arguments and literals in double angle brackets « ... » are arbitrary, i.e. they can appear any number – including zero – of times. Schema variables are denoted by &n. We skip Gegg-Harrison's notation for permutations of arguments and literals.

Schema A describes many well-known programs, e.g. *append/3*, *length/2*, *merge/3*, and *naive_reverse/2*. It is interesting to note that two of O'Keefe's schemata for recursive list processing programs [O'Keefe 90] – the *tower method* and the tail recursive *linear method* – are covered by schema A.

Schema B resembles schema A, but is doubly recursive. It is an abstraction of programs which rely on divide-and-conquer, e.g. *quicksort/2*, *mergesort/2*, and *flatten/2*.

The remaining twelve basic-level schemata describe other ways of processing the elements of a list, e.g. processing only a subset of the elements, or processing from the tail end. Details can be found in [Gegg-Harrison 89].

6 Transformation Schemata

Let us assume that a program to be transformed can be described abstractly by a program schema, and the transformed program by another program schema. In this

case, we may say that we transform an instance of one program schema into an instance of another one. This leads us to the idea to transform program schemata instead of programs, and to define *transformation schemata*. Transformation schemata are standardized transformations which transform input program schemata into output program schemata. Each transformation schema incorporates a fixed sequence of transformation steps, e.g. applications of unfold/fold rules. Transformation schemata incorporate our transformation decisions, i.e. the order and type of transformation steps, definitions of new predicates (eureka), selections of literals to be unfolded or folded, or any other information relevant to the transformation in question. In brief, each transformation schema represents one specific transformation strategy. Since the number of program schemata is limited we can – quasi at leisure – use all our intuition and our experience with individual program transformations to develop the transformation schemata.

Users of transformation schemata need not be concerned with individual transformation steps. Instead users can concentrate on the form of the input and output programs.

Let us make these ideas more concrete.

Transformation of a logic program means that a conjunction of literals

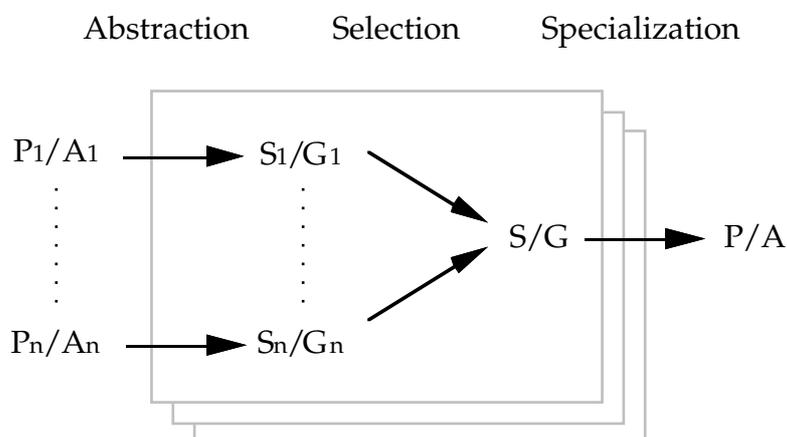
$$Q_1, A_1, \dots, A_n, Q_2$$

is replaced by another conjunction of literals

$$Q_1, A, Q_2$$

Each A_i calls a predicate P_i while A calls a predicate P which is semantically equivalent to the predicates P_1, \dots, P_n . Usually, we expect calculations using P to be more efficient than those using P_1, \dots, P_n .

For conciseness we say that we transform the set of terms $\{P_1/A_1, \dots, P_n/A_n\}$ into the term P/A . We derive P/A from $\{P_1/A_1, \dots, P_n/A_n\}$ by the three steps *abstraction*, *selection*, and *specialization*.



Abstraction

For each program P_i ($i=1, \dots, n$) we identify a program schema S_i which describes P_i abstractly. This abstraction generates a set of substitutions θ_i for schema variables. The same abstraction leads from the literals A_i to the abstract literals G_i . In short, the abstraction step replaces each term P_i/A_i by its abstraction S_i/G_i , and we have $P_i/A_i = S_i\theta_i/G_i\theta_i$.

Selection

Transformation schemata transform the set of abstract terms $\{S_1/G_1, \dots, S_n/G_n\}$ into an abstract term S/G , i.e. a transformation schema is defined as $\{S_1/G_1, \dots, S_n/G_n, S/G\}$. In general there are several transformation schemata which have $\{S_1/G_1, \dots, S_n/G_n\}$ as input. We select the transformation schema which generates a desired output program schema S together with an abstract literal G .

Specialization

We apply the substitution $\theta = \theta_1 \dots \theta_n$ to S/G to get the transformed program $P=S\theta$ and the transformed literal $A=G\theta$.

Once the transformation schemata are available these three steps can be automated to a large degree. User intervention is only necessary in the selection step.

7 A Transformation System Based on Transformation Schemata

Based on the preceding ideas we have implemented an experimental transformation system in Prolog.

We represent program schemata, e.g.

```
schema_A([], «&1»).
schema_A([H|T], «&2») :-
    <process1(«&3»),>
    schema_A(T, «&4»)
    <, process2(«&5») >.
```

as lists

```
[( Schema_A([], &1) :-
    true),
 ( Schema_A([H|T], &2) :-
    {process1},
    Schema_A(T, &4),
    {process2})]
```

Transformation schemata are similarly represented as facts *trafo/5*, e.g. the transformation schema *a1*. This schema can be used to transform a conjunction of two literals abstractly represented by G_1 and G_2 , and the two called programs represented by two copies S_1 and S_2 of program schema A into a literal represented by G and a program represented by S .

```

trafo(a1,
  [Schema_A1(L, &g1),                               % G1
   Schema_A2(L, &g2)],                               % G2
  [[(Schema_A1([], &t1) :-                            % S1
    true),
   (Schema_A1([H1 | T1], &t2) :-
    {process11},
    Schema_A1(T1, &t4),
    {process12})],
   [(Schema_A2([], &t1) :-                            % S2
    true),
   (Schema_A2([H2 | T2], &t2) :-
    {process21},
    Schema_A2(T2, &t4),
    {process22})]],
  [[Schema_A1, '_', Schema_A2, '_a1'](L, &g1, &g2)], % G
  [[(Schema_A1, '_', Schema_A2, '_a1')([], &t1, &t1) :- % S
    true),
   ((Schema_A1, '_', Schema_A2, '_a1')([H1 | T1], &t2, &t2) :-
    {process11},
    {process21},
    [Schema_A1, '_', Schema_A2, '_a1'](T1, &t4, &t4),
    {process12},
    {process22} )]).

```

This transformation schema is patterned after the transformation example of section 3. To derive S/G from S_1/G_1 and S_2/G_2 we have to perform the same transformation steps: one new definition, two unfolding steps, one folding step. This means that the transformation schema `a1` incorporates a specific transformation strategy for programs described by program schema `A`.

The top predicate *transform(Literals, NewLiterals, NewClauses)* of the transformation system transforms the conjunction *Literals* of literals into the conjunction *NewLiterals* and the clauses of the predicates called by *Literals* into *NewClauses*. The predicate *transform/3* is defined by

```

transform(Literals, NewLiterals, NewClauses) :-
  trafo(Trafo, InLiterals, InSchemata, OutLiterals, OutSchema),
  abstract(Literals, InLiterals, InSchemata, Bindings),
  select(Trafo),
  specialize(Bindings, OutLiterals, OutSchema, NewLiterals, NewClauses).

```

The goal *trafo/5* chooses a suitable transformation schema *Trafo*. The goal *select/1* asks the user to accept or to reject this choice. Rejection causes backtracking if there is more than one suitable transformation schema. The goals *abstract/4* and *specialize/5* implement the transformation steps with the same names.

8 More Example Transformations

Additional examples will show the flexibility of the approach and the power of the experimental transformation system.

First we want to revisit the *sum-length* example of section 3. The transformation schema *a1* transforms the conjunction of literals $sum(Xs,S),length(Xs,L)$ to the literal $sumlength(Xs,S,L)$ and derives at the the same time the program *sum_length_a1/3*..

```

sum(Xs, S), length(Xs, L)                % A1, A2
sum([], 0).                               % P1
sum([E | Es], S) :-
    sum(Es, S1),
    S is S1+E.

length([], 0).                             % P2
length([_ | Es], L) :-
    length(Es, L1),
    L is L1+1.

sum_length_a1([], 0, 0).                   % P
sum_length_a1([E | T], S, L) :-
    sum_length_a1(T, S1, L2),
    S is S1+E,
    L is L1+1.

sum_length_a1(Xs, S, L)                   % A

```

The same transformation schema *a1* can be used for the second example. Appending two lists by $append(L1,L2,L3)$ and subsequently calculating the length of the first list by $length(L1,N)$ involves two traversals of $L1$. We want to eliminate one traversal. The following transformation yields a program $append_length_a1(L1,L2,L3,N)$ which appends $L1$ and $L2$ and calculates at the same time the length N of $L1$.

```

append(L1, L2, L3), length(L1, N)        % A1, A2
append([], L, L).                         % P1
append([X | L1], L2, [X | L3]) :-
    append(L1, L2, L3).

length([], 0).                             % P2
length([_ | Es], N) :-
    length(Es, N1),
    N is N1+1.

append_length_a1([], L, L, 0).             % P
append_length_a1([X | L1], L2, [X | L3], N) :-
    append_length_a1(L1, L2, L3, N1),
    N is N1+1.

append_length_a1(L1, L2, L3, N)          % A

```

In the next example we append two lists by $append(L1,L2,L3)$ and calculate the length of the resulting list $L3$ by $length(L3,N)$. Appending $L1$ and $L2$ means traversing $L1$, while calculating the length of $L3$ means traversing $L3$ of which $L1$ is a prefix. As in the previous example, $L1$ is traversed twice, but for the calculation of the length of $L3$ the rest of $L3$ must also be traversed. We want to eliminate the unnecessary second traversal of $L1$. With the help of the transformation schema *a2*

```

trafo(a2,
  [Schema_A1(X, Y, Z, &g1),           % G1
  Schema_A2(Z, &g2)],                % G2
  [[(Schema_A1([], L11, L11, &11) :-   % S1
    true),
  (Schema_A1([H1 | T11], L12, [H1 | T12], &12) :-
    {process11},
    Schema_A1(T11, L12, T12, &14),
    {process12})],
  [(Schema_A2([], &21) :-             % S2
    true),
  (Schema_A2([H2 | T21], &22) :-
    {process21},
    Schema_A2(T21, &24),
    process22)]]],
  [[Schema_A1, '_', Schema_A2, '_a2'](X, Y, Z, &g1, &g2)], % G
  [[(Schema_A1, '_', Schema_A2, '_a2')([], [], [], &21) :- % S
    true),
  ((Schema_A1, '_', Schema_A2, '_a2')([], [H2 | T21], [H2 | T21], &11, &22) :-
    {process21},
    [Schema_A1, '_', Schema_A2, '_a2']([], T21, T21, &11, &24),
    {process22}),
  ((Schema_A1, '_', Schema_A2, '_a2')([H2 | T11], L, [H2 | T12], &12, &22) :-
    {process11},
    {process21},
    [Schema_A1, '_', Schema_A2, '_a2'](T11, L, T12, &14, &24),
    {process12},
    {process22})]].

```

we get the program *append_length_a2(L1,L2,L3,N)* which appends *L1* and *L2* and calculates at the same time the length *N* of the resulting list *L3*.

```

append(L1, L2, L3), length(L3, N)      % A1, A2
append([], L, L).                       % P1
append([X | L1], L2, [X | L3]) :-
  append(L1, L2, L3).
length([], 0).                           % P2
length([_ | Es], N) :-
  length(Es, N1),
  N is N1+1.
append_length_a2([], [], [], 0).         % P
append_length_a2([], [H | Es], [H | Es], N) :-
  append_length_a2([], Es, Es, N1),
  N is N1+1.
append_length_a2([X | L1], L2, [X | L3], N) :-
  append_length_a2(L1, L2, L3, N1),
  N is N1+1.
append_length_a2(L1, L2, L3, N)         % A

```

Both transformation schemata *a1* and *a2* are based on unfold/fold rules. The following example shows that transformation schemata can also be derived using other transformation rules. Recursive predicates can be made tail-recursive or iterative by

the introduction of an intermediate data structure called accumulator. The schema *t2* achieves this transformation.

```

trafo(t2,
  [Schema_T(L, R, &g1)],           % G1
  [[(Schema_T([], I, &1) :-
    true),
  (Schema_T([H|T], N, &2) :-
    {process1},
    Schema_T(T, N1, &4),
    Update(N1, HV, N, &5))]],
  [[Schema_T_t2(L, I, R, &g1)],     % G
  [[(Schema_T_t2([], RN, RN, &1) :-
    true),
  (Schema_T_t2([H|T], RN1, N, &2) :-
    {process1},
    Update(HV, RN1, N1, &5),
    [Schema_T_t2](T, N1, N, &4))]].

```

As a concrete example we transform the predicate *length/2* into its tail-recursive form *length_t2/3*. Note that the transformed literal *A* correctly initializes the accumulator to 0.

```

length(Xs, L)                       % A1
length([], 0).                       % P1
length([_ | T], N) :-
  length(T, N1),
  N is N1+1.

length_t2([], N, N).                 % P
length_t2([_ | T], N1, N) :-
  N2 is N1+1,
  length_t2(T, N2, N).

length_t2(Xs, 0, L)                  % A

```

Other transformation schemata have been implemented but will not be presented here because of the limited space.

9 Conclusions

There are several reasons why transformation schemata form a very powerful and flexible approach to program transformations.

Firstly, user interaction is enormously reduced and simplified. Instead of being confronted with the intricacies of the unfold/fold or other transformation rules, users can select a transformation by the very structure of the programs in question. If the transformation system offers a choice of possible transformation schemata, user interaction is reduced to yes/no decisions. For standard transformation situations one can even envisage eliminating user interaction altogether by heuristics.

Secondly, sets of transformation schemata can easily be extended and adapted by the addition of new schemata. The approach also allows to add transformation schemata for individual programs, especially for programs which do not fit into a given system of program schemata.

Thirdly, as demonstrated transformation schemata need not be derived from unfold/fold rules. Any set of semantics-preserving transformation rules can be used.

Fourthly, instead of deriving transformation schemata from transformation rules we could invent transformation schemata and post factum prove them correct. [Waldau 91] proposes to validate transformation schemata by formal proofs based on intuitionistic logic.

These and other extensions of the basic approach will be used to develop a comprehensive and practical transformation system for logic programs.

9 Acknowledgements

We would like to thank the anonymous referees for their helpful and valuable comments on an earlier version of the paper. One of us (NEF) profited from intensive discussions during the LOPSTR '91 workshop at the University of Manchester.

This research has been partially supported by the Swiss National Science Foundation under contract 2000-5.449.

10 References

- [Brna et al. 88] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, B. Smith, M. van Someren, Prolog Programming Techniques, DAI Research Paper, 403, Department of Artificial Intelligence, University of Edinburgh, 1988
- [Burstall & Darlington 77] R. Burstall, J. Darlington, Transformations for Developing Recursive Programs, JACM, Vol. 24, No. 1, 1977
- [Deville & Burnay 90] Y. Deville, J. Burnay, Generalization and Program Schemata. Proceedings of the North American Conference on Logic Programming 1989, E. L. Lusk, R. A. Overbeek (eds.), MIT Press, October 1989, pp. 409-425.
- [Gallagher 90] J. Gallagher, Program Analysis and Transformation, Handout at Logic Programming Summer School LPSS '90, University of Zurich, August 1990
- [Gardner & Shepherdson 89] P. A. Gardner, J. C. Shepherdson, Unfold/Fold Transformations of Logic Programs, Report PM-89-01, School of Mathematics, University of Bristol, 1989
- [Gegg-Harrison 89] T. S. Gegg-Harrison, Basic Prolog Schemata, CS-1989-20, Department of Computer Science, Duke University, September 1989

- [Lakhotia 89] A. Lakhotia, Incorporating Programming Techniques into Prolog Programs. Proceedings of the North American Conference on Logic Programming 1989, E. L. Lusk, R. A. Overbeek (eds.), MIT Press, October 1989, pp. 426-440.
- [Lakhotia & Sterling 88] A. Lakhotia, L. Sterling, How to Control Unfolding when Specializing Interpreters, in L. Sterling (ed.), The Practice of Prolog, MIT Press, 1990, pp. 171
- [Nielson & Nielson 90] H. R. Nielson, F. Nielson, Eureka Definitions for Free – Disagreement Points for Fold/Unfold Transformations, Proceedings of ESOP '90, LNCS 432, 1990, pp. 291-305
- [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 of ESOP '90, LNCS 432, 1990, pp. 306-325
- [Robertson 91] D. Robertson, A Simple Prolog Techniques Editor for Novice Users, Proceedings of ALPUK 91, 3rd Annual Conference on Logic Programming, Edinburgh, April 1991
- [Tamaki & Sato 84] H. Tamaki, T. Sato, Unfold/Fold Transformation of Logic Programs, Proceedings of the Second International Conference on Logic Programming, Uppsala, 1984, S. - Å. Tärnlund (ed.), University of Uppsala, pp. 127 - 138
- [Waldau 91] M. Waldau, Formal Validation of Transformation Schemas Using First-Order Proofs, Proceedings of LOPSTR '91, University of Manchester, Springer Workshop in Computing Series (this volume), 1991