Generalised Logic Program Transformation Schemas

Halime Büyükyıldız and Pierre Flener

Department of Computer Engineering and Information Science
Faculty of Engineering, Bilkent University, 06533, Bilkent, Ankara, Turkey
`pf@cs.bilkent.edu.tr`

Abstract. Schema-based logic program transformation has proven to be an effective technique for the optimisation of programs. This paper results from the research that began by investigating the suggestions in [11] to construct a more general database of transformation schemas for optimising logic programs at the declarative level. The proposed transformation schemas fully automate accumulator introduction (also known as descending computational generalisation), tupling generalisation (a special case of structural generalisation), and duality laws (which are extensions to relational programming of the first duality law of the fold operators in functional programming). The schemas are proven correct. A prototype schema-based transformation system is evaluated.

1 Introduction

Schema-based program *construction* and *synthesis* were studied in logic programming [9,10,16,14,23] and in functional programming [20,21]. Using schemas for logic program *transformation* was first studied in [13] and then extended in [25,18]. Schema-based logic program transformation was also studied in [11,15]. This paper results from the research that began by investigating the suggestions in [11] and extending the ideas in [1] to construct a database of more general transformation schemas for optimising logic programs at the declarative level. For full details of this research, the reader is invited to consult [5].

Throughout this paper, the word program (resp. procedure) is used to mean typed definite program (resp. procedure). An *open program* is a program where some of the relations appearing in the clause bodies are not appearing in any heads of clauses, and these relations are called *undefined* (or *open*) relations. If all the relations appearing in the program are *defined*, then the program is a *closed program.* The format of a *specification* S_r of a relation r is:

$$\forall X : \mathcal{X}.\ \forall Y : \mathcal{Y}.\ \ \mathcal{I}_r(X) \Rightarrow [r(X,Y) \Leftrightarrow \mathcal{O}_r(X,Y)]$$

where $\mathcal{I}_r(X)$ denotes the *input condition* that must be fulfilled before the execution of the procedure, and $\mathcal{O}_r(X,Y)$ denotes the *output condition* that will be fulfilled after the execution.

We now give the definitions of the notions that will be used throughout the paper. All the definitions are given for programs in closed *frameworks* [12]. A *framework* can be defined simply as a full first-order theory (with identity) with intended model. A closed framework has no parameters and open symbols. Thus, it completely defines an abstract data type (ADT).

Correctness and Equivalence Criteria. We first give correctness and equivalence criteria for programs.

Definition 1 ((Correctness of a Closed Program)).
Let P be a closed program for a relation r in a closed framework $\mathcal{F}$. We say that P is (*totally*) *correct* wrt its specification S_r iff, for any ground term t of $\mathcal{X}$ such that $\mathcal{I}_r(t)$ holds, we have $P \vdash r(t, u)$ iff $\mathcal{F} \models \mathcal{O}_r(t, u)$, for every ground term u of $\mathcal{Y}$. If we replace 'iff' by 'implies' in the condition above, then P is said to be *partially correct* wrt S_r, and if we replace 'iff' by 'if', then P is said to be *complete* wrt S_r.

This kind of correctness is not entirely satisfactory, for two reasons. First, it defines the correctness of P in terms of the procedures for the relations in its clause bodies, rather than in terms of their specifications. Second, P must be a closed program, even though it might be desirable to discuss the correctness of P without having to fully implement it. So, the abstraction achieved through the introduction (and specification) of the relations in its clause bodies is wasted. This leads us to the notion of steadfastness (also known as parametric correctness) [12,9].

Definition 2 ((Steadfastness of an Open Program)).
In a closed framework $\mathcal{F}$, let:

– P be an open program for a relation r
– $q_1, \ldots, q_m$ be all the undefined relation names appearing in P
– $S_1, \ldots, S_m$ be the specifications of $q_1, \ldots, q_m$.

We say that P is *steadfast* wrt its specification S_r in $\{S_1, \ldots, S_m\}$ iff the (closed) program $P \cup P_S$ is correct wrt S_r, where P_S is any closed program such that:

– P_S is correct wrt each specification S_j $(1 \leq j \leq m)$
– P_S contains no occurrences of the relations defined in P.

For program equivalence, now, we do not require the two programs to have the same models, because this would not make much sense in some program transformation settings, where the transformed program features relations that are not in the initially given program. That is why our program equivalence criterion establishes equivalence wrt the specification of a common relation (usually the root of their call-hierarchies).

Definition 3 ((Equivalence of Two Open Programs)).
In a closed framework $\mathcal{F}$, let P and Q be two open programs for a relation r. Let $S_1, \ldots, S_m$ be the specifications of $p_1, \ldots, p_m$, which are all the undefined relation names appearing in P, and let $S'_1, \ldots, S'_t$ be the specifications of

q_1,..., q_t, which are all the undefined relation names appearing in Q. We say that $\langle P, \{S_1, \ldots, S_m\}\rangle$ is *equivalent to* $\langle Q, \{S'_1, \ldots, S'_t\}\rangle$ wrt the specification S_r (or simply that P is equivalent to Q wrt S_r) when P is steadfast wrt S_r in $\{S_1, \ldots, S_m\}$ and Q is steadfast wrt S_r in $\{S'_1, \ldots, S'_t\}$. Since the 'is equivalent to' relation is symmetric, we also say that P and Q are *equivalent* wrt S_r.

In program transformation settings, there sometimes are conditions that have to be satisfied by some parts of the initial and/or transformed program in order to have a transformed program that is equivalent to the initially given program wrt the specification of the top-level relation. Hence the following definition.

Definition 4 ((Conditional Equivalence of Two Open Programs)).

In a closed framework $\mathcal{F}$, let P and Q be two open programs for a relation r. We say that P is *equivalent to* Q wrt the specification S_r *under* conditions C iff P is *equivalent to* Q wrt S_r whenever C holds.

Program Schemas and Schema Patterns. The notion of program schema was also used in [9,10,11,13,16,14,15,6,25], but here we have an additional component, which makes our definition [12] of program schemas different.

Definition 5 ((Program Schemas)).
In a closed framework $\mathcal{F}$, a *program schema* for a relation r is a pair $\langle T, C\rangle$, where T is an open program for r, called the *template*, and C is a set of specifications of the open relations of T, in terms of each other and in terms of the input/output conditions of the closed relations of T. The specifications in C, called the *steadfastness constraints*, are such that, in $\mathcal{F}$, T is steadfast wrt its specification S_r in C.

Sometimes, a series of schemas are quite similar, in the sense that they only differ in the number of arguments of some relation, or in the number of calls to some relation, etc. For this purpose, rather than having a proliferation of similar schemas, we introduce the notion of *schema pattern* (compare with [6]).

Definition 6 ((Schema Patterns)).
A *schema pattern* is a program schema where term, conjunct, and disjunct ellipses are allowed in the template and in the steadfastness constraints.

For instance, $TX_1, \ldots, TX_t$ is a term ellipsis, and $\bigwedge_{i=1}^{t} r(TX_i, TY_i)$ is a conjunct ellipsis. Our schemas are more general than those in [11] in the sense that we now allow such ellipses.

Schema-based Program Transformation. In schema-based transformation, transformation techniques are pre-compiled at the schema-level.

Definition 7 ((Transformation Schemas)).
A *transformation schema* is a 5-tuple $\langle S_1, S_2, A, O_{12}, O_{21}\rangle$, where S_1 and S_2 are program schemas (or schema patterns), A is a set of *applicability conditions*,

which ensure the equivalence of the templates of S_1 and S_2 wrt the specification of the top-level relation, and O_{12} (resp. O_{21}) is a set of *optimisability conditions*, which ensure further optimisability of the output program schema (or schema pattern) S_2 (resp. S_1).

If a transformation schema embodies some generalisation technique, then it is called a *generalisation schema*. The problem generalisation techniques that are used in this paper are explained in detail in [9]. Using these techniques for synthesising and/or transforming a program in a schema-based fashion was first proposed in [9,10], and then extended in [11]. The generalisation methods that we pre-compile in our transformation schemas are *tupling generalisation*, which is a special case of *structural generalisation* where the structure of some parameter is generalised, and *descending generalisation*, which is a special case of *computational generalisation* where the general state of computation is generalised in terms of what remains to be done. If a transformation schema embodies a duality law, then it is called a *duality schema*.

In the remainder of this paper, we first give two divide-and-conquer schema patterns in Section 2. We then explain in detail how automation of program transformation is achieved by tupling and descending generalisation, in Sections 3 and 4. In Section 5, we explain the duality schemas. In Section 6, we discuss, by using the results of performance tests, the effects of the optimisability conditions in the transformation schemas. Before we conclude, the prototype transformation system, which was developed to test the practicality of the ideas explained in this paper, is presented in Section 7.

2 Divide-and-Conquer Programs

The schema patterns in this section abstract sub-families of divide-and-conquer (DC) programs. They are here restricted to binary relations with X as the induction parameter and Y as the result parameter, to reflect the schema patterns that can be handled by the prototype transformation system explained in Section 7. Another restriction in the schema patterns is that when X is non-minimal, then X is decomposed into $h = 1$ head HX and $t > 0$ tails $TX_1, \ldots, TX_t$, so that Y is composed from 1 head HY (which is the result of processing HX) and t tails $TY_1, \ldots, TY_t$ (which are the results of recursively calling the relation with $TX_1, \ldots, TX_t$, respectively) by p-fix composition (i.e., Y is composed by putting its head HY between its tails TY_{p-1} and TY_p).

These schema patterns are called $DCLR$ and $DCRL$ (the reason for these names will be explained soon). Template $DCLR$ (resp. $DCRL$) below is the template of the $DCLR$ (resp. $DCRL$) schema pattern. In these patterns, *minimal*, *solve*, etc., denote place-holders for relation symbols. During the particularisation of a schema pattern to a schema, all these place-holders are *renamed*, because otherwise all divide-and-conquer programs would have the same relation symbols. Indeed, since a template is an open program, the idea is to obtain concrete programs from the template by *adding* programs for the open relations, such that these programs satisfy the steadfastness constraints. The steadfastness constraints corresponding to these DC templates (i.e., the specifications of their open relations) are the same, since these templates have the same open relations. Such constraints are shown in [12] in this volume.

$r(X, Y) \leftarrow$
$\quad minimal(X),$
$\quad solve(X, Y)$
$r(X, Y) \leftarrow$
$\quad nonMinimal(X),$
$\quad decompose(X, HX, TX_1, \ldots, TX_t),$
$\quad r(TX_1, TY_1), \ldots, r(TX_t, TY_t),$
$\quad init(I_0),$
$\quad compose(I_0, TY_1, I_1), \ldots, compose(I_{p-2}, TY_{p-1}, I_{p-1}),$
$\quad process(HX, HY), compose(I_{p-1}, HY, I_p),$
$\quad compose(I_p, TY_p, I_{p+1}), \ldots, compose(I_t, TY_t, I_{t+1}),$
$\quad Y = I_{t+1}$

Template *DCLR*

$r(X, Y) \leftarrow$
$\quad minimal(X),$
$\quad solve(X, Y)$
$r(X, Y) \leftarrow$
$\quad nonMinimal(X),$
$\quad decompose(X, HX, TX_1, \ldots, TX_t),$
$\quad r(TX_1, TY_1), \ldots, r(TX_t, TY_t),$
$\quad init(I_{t+1}),$
$\quad compose(TY_t, I_{t+1}, I_t), \ldots, compose(TY_p, I_{p+1}, I_p),$
$\quad process(HX, HY), compose(HY, I_p, I_{p-1}),$
$\quad compose(TY_{p-1}, I_{p-1}, I_{p-2}), \ldots, compose(TY_1, I_1, I_0),$
$\quad Y = I_0$

Template *DCRL*

We can now explain the underlying idea why we have two different schema patterns for DC, and why we call them *DCLR* and *DCRL*. If we denote the functional version of the *compose* relation with $\oplus$, then the composition of Y in template *DCLR* by *left-to-right* (*LR*) *composition ordering* can be written as:

$$Y = ((((((e \oplus TY_1) \oplus \ldots) \oplus TY_{p-1}) \oplus HY) \oplus TY_p) \oplus \ldots) \oplus TY_t \qquad (1)$$

where e is the (unique) term for which *init* holds. Similarly, the composition of Y in *DCRL* by *right-to-left* (*RL*) *composition ordering* can be written as:

$$Y = TY_1 \oplus (\ldots \oplus (TY_{p-1} \oplus (HY \oplus (TY_p \oplus (\ldots \oplus (TY_t \oplus e)))))) \qquad (2)$$

Throughout the paper, we use the *infix_flat* problem, whose DC programs are given in the example below.

Example 1. The specification of *infix_flat* is:

$infix\_flat(B, F)$ iff list F is the infix representation of binary tree B

where *infix representation* means the list representation of the infix traversal of the tree. Program 1 (resp. Program 2) below is the program for the $infix\_flat/2$ problem that is a (partially evaluated) instance of the *DCLR* (resp. *DCRL*) schema pattern, for $t = p = 2$. Note the line-by-line correspondence between the program computations and the templates.

```
infix_flat(B, F) ←
        B = void,
        F = []
infix_flat(B, F) ←
        B = bt(_, _, _),
        B = bt(L, E, R),
        infix_flat(L, FL),
                infix_flat(R, FR),
        I0 = [],
        append(I0, FL, I1),
        HF = [E], append(I1, HF, I2),
        append(I2, FR, I3),
        F = I3
```

Program 1

```
infix_flat(B, F) ←
        B = void,
        F = []
infix_flat(B, F) ←
        B = bt(_, _, _),
        B = bt(L, E, R),
        infix_flat(L, FL),
                infix_flat(R, FR),
        I3 = [],
        append(FR, I3, I2),
        HF = [E], append(HF, I2, I1),
        append(FL, I1, I0),
        F = I0
```

Program 2

3 Program Transformation by Tupling Generalisation

If a program for a relation r, which has the specification S_r of Section 1, is given as an instance of *DCLR* (or *DCRL*), then the specification of the tupling-generalised problem of r, namely $S_{r\_tupling}$, is:

$$\begin{gathered}\forall Xs : \textit{list of } \mathcal{X}.\ \forall Y : \mathcal{Y}.\ \ (\forall X : \mathcal{X}.\ X \in Xs \Rightarrow \mathcal{I}_r(X)) \Rightarrow \\ (r\_tupling(Xs, Y) \Leftrightarrow (Xs = [\,] \wedge Y = e) \vee (Xs = [X_1, X_2, \ldots, X_n] \\ \wedge \bigwedge_{i=1}^{n} \mathcal{O}_r(X_i, Y_i)\ \wedge I_1 = Y_1 \wedge \bigwedge_{i=2}^{n} \mathcal{O}_c(I_{i-1}, Y_i, I_i)\ \wedge Y = I_n))\end{gathered}$$

where $\mathcal{O}_c$ is the output condition of *compose*, and e is the (unique) term for which *init* holds.

The tupling generalisation schemas (one for each DC schema pattern) are:

TG_1 : $\langle$ $DCLR$, TG, A_{t_1}, $O_{t_1 12}$, $O_{t_1 21}$ $\rangle$ where

A_{t_1} : - *compose* is associative
- *compose* has e as the left and right identity element
- $\forall X : \mathcal{X}.\ \ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

$O_{t_1 12}$: partial evaluation of the conjunction $process(HX, HY), compose(HY, TY, Y)$ results in the introduction of a non-recursively defined relation

$O_{t_1 21}$: partial evaluation of the conjunction $process(HX, HY), compose(I_{p-1}, HY, I_p)$ results in the introduction of a non-recursively defined relation

TG_2 : $\langle$ $DCRL$, TG, A_{t_2}, $O_{t_2 12}$, $O_{t_2 21}$ $\rangle$ where

A_{t_2} : - *compose* is associative
- *compose* has e as the left and right identity element
- $\forall X : \mathcal{X}.\ \ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

$O_{t_2 12}$: partial evaluation of the conjunction $process(HX, HY), compose(HY, TY, Y)$ results in the introduction of a non-recursively defined relation

$O_{t_2 21}$: partial evaluation of the conjunction $process(HX, HY), compose(HY, I_p, I_{p-1})$ results in the introduction of a non-recursively defined relation

where the template of the common schema pattern TG is given on the next page.

Note that, in the TG template, *all* the open relations of $DCLR$ (or $DCRL$) appear, but *no* new relations. This crucial observation enables the once-and-for-all verification of the conditional equivalence of the two templates in the TG_i transformation schemas wrt S_r under A_{t_i}, which verification is thus independent of the actual specifications of the open relations (i.e., the steadfastness constraints) [4].

The applicability conditions of TG_1 (resp. TG_2) ensure the equivalence of the $DCLR$ (resp. $DCRL$) and TG programs for a given problem. The optimisability conditions ensure that the output programs of these generalisation schemas can be made more efficient than the input programs. Indeed, the optimisability conditions, together with some of the applicability conditions, check whether the *compose* calls in the TG template can be eliminated. For instance, the conjunction $solve(X, HY), compose(HY, TY, Y)$ in the second clause of *r_tupling* can be simplified to $Y = A$, if relation r maps the minimal form of X into e, and if e is also the right identity element of *compose*. This is already checked by the second and third applicability conditions of TG_1 and TG_2. Also, in the third and fourth clauses of *r_tupling*, the conjunction $process(HX, HY), compose(HY, TY, Y)$ can be partially evaluated, resulting in the disappearance of that call to *compose*, and thus in a merging of the *compose* loop into the r loop in the $DCLR$ (or $DCRL$) template, if the optimisability condition $O_{t_1 12}$ (or $O_{t_2 12}$) holds.

Let us illustrate tupling generalisation by applying the TG_i generalisation schemas on the $infix\_flat$ problem.

```
r(X, Y) ←
  r_tupling([X], Y)
r_tupling(Xs, Y) ←
  Xs = [ ],
  init(Y)
r_tupling(Xs, Y) ←
  Xs = [X|TXs],
  minimal(X),
  r_tupling(TXs, TY),
  solve(X, HY),
  compose(HY, TY, Y),
r_tupling(Xs, Y) ←
  Xs = [X|TXs],
  nonMinimal(X),
  decompose(X, HX, TX_1, ..., TX_t),
  minimal(TX_1), ..., minimal(TX_t),
  r_tupling(TXs, TY),
  process(HX, HY),
  compose(HY, TY, Y)
r_tupling(Xs, Y) ←
  Xs = [X|TXs],
  nonMinimal(X),
  decompose(X, HX, TX_1, ..., TX_t),
  minimal(TX_1), ..., minimal(TX_{p-1}),
  (nonMinimal(TX_p); ... ; nonMinimal(TX_t)),
  r_tupling([TX_p, ..., TX_t|TXs], TY),
  process(HX, HY),
  compose(HY, TY, Y)
r_tupling(Xs, Y) ←
  Xs = [X|TXs],
  nonMinimal(X),
  decompose(X, HX, TX_1, ..., TX_t),
  (nonMinimal(TX_1); ... ; nonMinimal(TX_{p-1})),
  minimal(TX_p), ..., minimal(TX_t),
  minimal(U_1), ..., minimal(U_{p-1}),
  decompose(N, HX, U_1, ..., U_{p-1}, TX_p, ..., TX_t),
  r_tupling([TX_1, ..., TX_{p-1}, N|TXs], Y)
r_tupling(Xs, Y) ←
  Xs = [X|TXs],
  nonMinimal(X),
  decompose(X, HX, TX_1, ..., TX_t),
  (nonMinimal(TX_1); ... ; nonMinimal(TX_{p-1})),
  (nonMinimal(TX_p); ... ; nonMinimal(TX_t)),
  minimal(U_1), ..., minimal(U_t),
  decompose(N, HX, U_1, ..., U_t),
  r_tupling([TX_1, ..., TX_{p-1},
          N, TX_p, ..., TX_t|TXs], Y)
```

```
infix_flat(B, F) ←
  infix_flat_t([B], F)
infix_flat_t(Bs, F) ←
  Bs = [ ],
  F = [ ]
infix_flat_t(Bs, F) ←
  Bs = [B|TBs],
  B = void,
  infix_flat_t(TBs, TF),
  HF = [ ],
  append(HF, TF, F)
infix_flat_t(Bs, F) ←
  Bs = [B|TBs],
  B = bt(_, _, _),
  B = bt(L, E, R),
  L = void, R = void,
  infix_flat_t(TBs, TF),
  HF = [E],
  append(HF, TF, F)
infix_flat_t(Bs, F) ←
  Bs = [B|TBs],
  B = bt(_, _, _),
  B = bt(L, E, R),
  L = void,
  R = bt(_, _, _),
  infix_flat_t([R|TBs], TF),
  HF = [E],
  append(HF, TF, F)
infix_flat_t(Bs, F) ←
  Bs = [B|TBs],
  B = bt(_, _, _),
  B = bt(L, E, R),
  L = bt(_, _, _),
  R = void,
  U_L = void,
  N = bt(U_L, E, R),
  infix_flat_t([L, N|TBs], TF),
infix_flat_t(Bs, F) ←
  Bs = [B|TBs],
  B = bt(_, _, _),
  B = bt(L, E, R),
  L = bt(_, _, _),
  R = bt(_, _, _),
  U_L = void, U_R = void,
  N = bt(U_L, E, U_R),
  infix_flat_t([L, N, R|TBs], F)
```

Template TG and Program 3

Example 2. The specification of the $infix\_flat$ problem, and its $DCLR$ and $DCRL$ programs, are in Example 1 in Section 2. The $infix\_flat$ problem can be tupling-generalised using the TG_i transformation schemas above, since the $infix\_flat$ programs have open relations that satisfy the applicability and optimisability conditions of these schemas. So, the specification of the tupling-generalised problem of $infix\_flat$ is:

$infix\_flat\_t(Bs, F)$ iff F is the concatenation of the infix representations of the elements in the binary tree list Bs.

Program 3 on the previous page is the tupling-generalised program for $infix\_flat$ as an instance of TG, for $t = p = 2$.

Although the tupling generalisation schemas are constructed to tupling-generalise DC programs (i.e., to transform DC programs into TG programs), these schemas can also be used in the reverse direction, such that they transform TG programs into DC programs, provided the optimisability conditions for the corresponding DC schema pattern are satisfied; note that applicability conditions work in both directions.

4 Program Transformation by Descending Generalisation

Descending generalisation [9] can also be called the *accumulation strategy* (as in functional programming [2], and in logic programming [17]), since it introduces an accumulator parameter and progressively extends it to the final result. Descending generalisation can also be seen as transformation towards *difference structure* manipulation, since any form of difference structures can be created by descending generalisation, and not just difference-lists.

Four descending generalisation schemas (two for each DC schema pattern) are given. Since the applicability conditions of each descending generalisation schema are different, the process of choosing the appropriate generalisation schema for the input DC program is done only by checking the applicability and optimisability conditions, and the eureka (i.e., the specification of the generalised problem) then comes for free.

The reason why we call the descendingly generalised (DG) schema patterns '$DGLR$' and '$DGRL$' is similar to the reason why we call the divide-and-conquer schema patterns $DCLR$ and $DCRL$, respectively. In descending generalisation, the composition ordering for extending the accumulator parameter in the template $DGLR$ (resp. $DGRL$) is from *left-to-right* (LR) (resp. *right-to-left* (RL)).

The first two descending generalisation schemas are:

DG_1 : $\langle$ $DCLR$, $DGLR$, A_{dg_1}, $O_{dg_1 12}$, $O_{dg_1 21}$ $\rangle$ where
  A_{dg_1} : - *compose* is associative
    - *compose* has e as the left identity element
  $O_{dg_1 12}$: - *compose* has e as the right identity element
    and $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
    - partial evaluation of the conjunction

$process(HX, HY), compose(A_{p-1}, HY, A_p)$
results in the introduction of a non-recursively defined relation

$O_{dg_1 21}$: - partial evaluation of the conjunction
$process(HX, HY), compose(I_{p-1}, HY, I_p)$
results in the introduction of a non-recursively defined relation

DG_4 : $\langle$ *DCRL*, *DGLR*, A_{dg_4}, $O_{dg_4 12}$, $O_{dg_4 21}\rangle$ where
A_{dg_4} : - *compose* is associative
- *compose* has e as the left and right identity element
$O_{dg_4 12}$: - $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(A_{p-1}, HY, A_p)$
results in the introduction of a non-recursively defined relation
$O_{dg_4 21}$: - partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursively defined relation

where e is the (unique) term for which *init* holds. These schemas have the *same* formal specification (i.e., eureka) for the relation $r\_descending_1$ of the schema pattern *DGLR*, namely:

$$\forall X : \mathcal{X}.\ \forall Y, A : \mathcal{Y}.\ \ \mathcal{I}_r(X) \Rightarrow$$
$$[r\_descending_1(X, Y, A) \ \Leftrightarrow\ \exists S : \mathcal{Y}.\ \mathcal{O}_r(X, S) \wedge \mathcal{O}_c(A, S, Y)]$$

where $\mathcal{O}_c$ is the output condition of *compose*. The template of the common schema pattern *DGLR* of DG_1 and DG_4 is:

```
$r(X, Y) \leftarrow$
  $init(A), r\_descending_1(X, Y, A)$
$r\_descending_1(X, Y, A) \leftarrow$
  $minimal(X),$
  $solve(X, S), compose(A, S, Y)$
$r\_descending_1(X, Y, A) \leftarrow$
  $nonMinimal(X),$
  $decompose(X, HX, TX_1, \ldots, TX_t),$
  $init(E), compose(A, E, A_0),$
  $r\_descending_1(TX_1, A_1, A_0), \ldots, r\_descending_1(TX_{p-1}, A_{p-1}, A_{p-2}),$
  $process(HX, HY), compose(A_{p-1}, HY, A_p),$
  $r\_descending_1(TX_p, A_{p+1}, A_p), \ldots, r\_descending_1(TX_t, A_{t+1}, A_t),$
  $Y = A_{t+1}$
```

Template *DGLR*

Note that, in the *DGLR* template, *all* the open relations of *DCLR* (or *DCRL*) appear, but *no* new relations. The applicability and optimisability conditions of these two generalisation schemas differ, since the composition ordering is changed from RL to LR in DG_4.

We now illustrate descending generalisation on our $infix\_flat$ problem.

Example 3. The specification of a program for the LR descendingly generalised version of $infix\_flat$ is:

$infix\_flat\_descending_1(B, F, A)$ iff list F is the concatenation of list A and the infix representation of binary tree B.

Program 4 is the program for $infix\_flat$ as an instance of $DGLR$, for $t = p = 2$.

```
$infix\_flat(B, F) \leftarrow$
  $infix\_flat\_descending_1(B, F, [\,])$
$infix\_flat\_descending_1(B, F, A) \leftarrow$
  $B = void,$
  $S = [\,], append(A, S, F)$
$infix\_flat\_descending_1(B, F, A) \leftarrow$
  $B = bt(\_, \_, \_),$
  $B = bt(L, E, R),$
  $append(A, [\,], A_0),$
  $infix\_flat\_descending_1(L, A_1, A_0),$
  $HF = [E], append(A_1, HF, A_2),$
  $infix\_flat\_descending_1(R, A_3, A_2),$
  $F = A_3$
```

Program 4

Since the applicability conditions of DG_1 (resp. DG_4) are satisfied for the input $DCLR$ (resp. $DCRL$) $infix\_flat$ program, the descendingly generalised $infix\_flat$ program can be Program 4. However, for this problem, descending generalisation of the $infix\_flat$ programs with the DG transformation schemas above should *not* be done, since the optimisability conditions of DG_1 (resp. DG_4) are not satisfied by the open relations of $infix\_flat$. Indeed, in the non-minimal case of $infix\_flat\_descending_1$, partial evaluation of the conjunction $HF = [E], append(A_1, HF, A_2)$ does not result in the introduction of a non-recursively defined relation, because of properties of *append* (actually, due to the inductive definition of lists). Moreover, the induction parameter of *append*, which is here the accumulator parameter, increases in length each time *append* is called in the non-minimal case, which shows that this program is not a good choice as a descendingly generalised program for this problem. So, the optimisability conditions are really useful to prevent non-optimising transformations.

The other two descending generalisation schemas are:

DG_2 : $\langle$ $DCLR$, $DGRL$, A_{dg_2}, $O_{dg_2 12}$, $O_{dg_2 21}$ $\rangle$ where
  A_{dg_2} : - *compose* is associative
    - *compose* has e as the left and right identity element
  $O_{dg_2 12}$: - $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
    - partial evaluation of the conjunction

$process(HX, HY), compose(HY, A_p, A_{p-1})$
results in the introduction of a non-recursively defined relation
$O_{dg_2 21}$: - partial evaluation of the conjunction
$process(HX, HY), compose(I_{p-1}, HY, I_p)$
results in the introduction of a non-recursively defined relation

DG_3 : $\langle$ *DCRL*, *DGRL*, A_{dg_3}, $O_{dg_3 12}$, $O_{dg_3 21}\rangle$ where
A_{dg_3} : - *compose* is associative
- *compose* has e as the right identity element
$O_{dg_3 12}$: - *compose* has e as the left identity element
and $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(HY, A_p, A_{p-1})$
results in the introduction of a non-recursively defined relation
$O_{dg_3 21}$: - partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursively defined relation

where e is the (unique) term for which *init* holds. These schemas have the *same* formal specification (i.e., eureka) for the relation *r_descending*$_2$ of the schema pattern *DGRL*, namely:

$$\forall X : \mathcal{X}.\ \forall Y, A : \mathcal{Y}.\ \ \mathcal{I}_r(X) \Rightarrow$$
$$[r\_descending_2(X, Y, A)\ \Leftrightarrow\ \exists S : \mathcal{Y}.\ \mathcal{O}_r(X, S) \wedge \mathcal{O}_c(S, A, Y)]$$

where $\mathcal{O}_c$ is the output condition of *compose*. Note the reversal of the roles of A and S compared to the specification of *r_descending*$_1$ above. The template of the common schema pattern *DGRL* of DG_2 and DG_3 is:

$r(X, Y) \leftarrow$
  $init(A), r\_descending_2(X, Y, A)$
$r\_descending_2(X, Y, A) \leftarrow$
  $minimal(X),$
  $solve(X, S), compose(S, A, Y)$
$r\_descending_2(X, Y, A) \leftarrow$
  $nonMinimal(X),$
  $decompose(X, HX, TX_1, \ldots, TX_t),$
  $init(E), compose(E, A, A_{t+1}),$
  $r\_descending_2(TX_t, A_t, A_{t+1}), \ldots, r\_descending_2(TX_p, A_p, A_{p+1}),$
  $process(HX, HY), compose(HY, A_p, A_{p-1}),$
  $r\_descending_2(TX_{p-1}, A_{p-2}, A_{p-1}), \ldots, r\_descending_2(TX_1, A_0, A_1),$
  $Y = A_0$

Template *DGRL*

Note that, in the *DGRL* template, *all* the open relations of *DCLR* (or *DCRL*) appear, but *no* new relations. The applicability and optimisability conditions of these two generalisation schemas differ, since the composition ordering is changed from LR to RL in DG_2.

Example 4. The specification of a program for the RL descendingly generalised version of $infix\_flat$ is:

$infix\_flat\_descending_2(B, F, A)$ iff list F is the concatenation of the infix representation of binary tree B and list A.

Program 5 is the program for $infix\_flat$ as an instance of $DGRL$, for $t = p = 2$.

```
$infix\_flat(B, F) \leftarrow$
  $infix\_flat\_descending_2(B, F, [\,])$
$infix\_flat\_descending_2(B, F, A) \leftarrow$
  $B = void,$
  $S = [\,], append(S, A, F)$
$infix\_flat\_descending_2(B, F, A) \leftarrow$
  $B = bt(\_, \_, \_),$
  $B = bt(L, E, R),$
  $append([], A, A_3),$
  $infix\_flat\_descending_2(R, A_2, A_3),$
  $HF = [E], append(HF, A_2, A_1),$
  $infix\_flat\_descending_2(L, A_0, A_1),$
  $F = A_0$
```

Program 5

Since both the applicability conditions and the optimisability conditions of DG_2 (resp. DG_3) are satisfied for the input $DCLR$ (resp. $DCRL$) $infix\_flat$ program, descending generalisation of the $infix\_flat$ programs results in Program 5. Partial evaluation of the conjunction $HF = [E], append(HF, A_2, A_1)$ in the non-minimal case of $infix\_flat\_descending_2$ then results in $A_1 = [E|A_2]$. Similarly, partial evaluation of the conjunction $S = [\,], append(S, A, F)$ in the minimal case results in $F = A$. Altogether, this amounts to the elimination of *append.*

Although the descending generalisation schemas are constructed to descendingly generalise DC programs, these schemas can also be used to transform DG programs into DC programs, provided the optimisability conditions for the corresponding DC schema pattern are satisfied. It is thus possible that we have Program 4 for the $infix\_flat$ problem, and that we want to transform it into a more efficient program; then the DC programs are good candidates, if we have the descending generalisation schemas above.

5 Program Transformation Using Duality Laws

In Section 2, while we discussed the composition ordering in the DC program schemas, the reader who is familiar with functional programming has noticed the similarities with the *fold* operators in functional programming. A detailed explanation of the fold operators and their laws can be found in [3]. Here, we only give the definitions of the *fold* operators, and their first law. The *foldr* operator can be defined as follows:

$$foldr\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus a) \ldots))$$

where $\oplus$ is a variable that is bound to a function of two arguments.
Similarly, the *foldl* operator can be defined as follows:

$$foldl\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = (\ldots((a \oplus x_1) \oplus x_2) \ldots) \oplus x_n$$

Thus, equation (1) in Section 2, which illustrates the composition of Y in the $DCLR$ template, can be rewritten using *foldl*:

$$Y = foldl\ (\oplus)\ e\ [TY_1, \ldots, TY_{p-1}, HY, TY_p, \ldots, TY_t]$$

Similarly, the *foldr* operator can be used to rewrite equation (2), which illustrates the composition of Y in the $DCRL$ template:

$$Y = foldr\ (\oplus)\ e\ [TY_1, \ldots, TY_{p-1}, HY, TY_p, \ldots, TY_t]$$

The first three laws of the fold operators are called *duality theorems*. The *first duality theorem* states that:

$$foldr\ (\oplus)\ a\ xs = foldl\ (\oplus)\ a\ xs$$

if $\oplus$ is associative and has (left/right) identity element a, and xs is a finite list.

By adding optimisability conditions, we can now devise a transformation schema based on this first duality theorem (compare with [16]):

D_{dc} : $\langle$ *DCLR*, *DCRL*, A_{ddc}, O_{ddc12}, $O_{ddc21}\rangle$ where
  A_{ddc} : - *compose* is associative
    - *compose* has e as the left and right identity element
  O_{ddc12} : - partial evaluation of the conjunction
    $process(HX, HY), compose(HY, I_p, I_{p-1})$
    results in the introduction of a non-recursively defined relation
  O_{ddc21} : - partial evaluation of the conjunction
    $process(HX, HY), compose(I_{p-1}, HY, I_p)$
    results in the introduction of a non-recursively defined relation

where e is the (unique) term for which *init* holds, where the schema patterns $DCLR$ and $DCRL$ are given in Section 2, and where A_{ddc} comes from the constraints of the first duality theorem. The optimisability conditions check whether the *compose* operator can be eliminated in the output program.

Similarly, it is possible to give a duality schema between the *DG* schema patterns:

D_{dg} : $\langle$ *DGLR*, *DGRL*, A_{ddg}, O_{ddg12}, $O_{ddg21}\rangle$ where
  A_{ddg} : - *compose* is associative
    - *compose* has e as the left and right identity element

O_{ddg12}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(HY, A_p, A_{p-1})$
results in the introduction of a non-recursively defined relation

O_{ddg21}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(A_{p-1}, HY, A_p)$
results in the introduction of a non-recursively defined relation

where e is the (unique) term for which *init* holds, and where the schema patterns *DGLR* and *DGRL* are given in Section 4.

6 Evaluation of the Transformation Schemas

We evaluate the transformation schemas using performance tests done on partially evaluated input and output programs of each transformation schema. However, the reader may find this evaluation a little bit dubious, since the transformation schemas in this paper are only dealing with the declarative features of programs. This evaluation is made because we think that these performance tests will help us see what our theoretical results amount to when tested practically, although in an environment with procedural side-effects. The programs are executed using Mercury 0.6 (for an overview of Mercury, please refer to [22]) on a SPARCstation 4. Since the programs are really short, the procedures were called 500 or 1000 times to achieve meaningful timing results. In Table 1, the results of the performance tests for five selected relations are shown, where each column heading represents the schema pattern to which the program written for the relation of that row belongs. (Of course, *quicksort* is not really a relation: we just mean to indicate that some partitioning is used as *decompose* for the *sort* relation.) The timing results are normalised wrt the DCLR column.

relations	DCLR	DCRL	TG	DGLR	DGRL
Prefix *flat*	1.00	0.92	0.23	11.88	0.15
Infix *flat*	1.00	0.49	0.02	7.78	0.05
Postfix *flat*	1.00	0.69	0.14	5.48	0.09
reverse	1.00	1.00	0.04	1.01	0.01
quicksort	1.00	0.85	0.72	6.02	0.56

Table 1. Performance test results

The reason why we chose the relations above is that for all the five considered schema patterns programs can be written for these relations.

Let us first compare the *DCLR* and *DCRL* schema patterns. For *reverse*, the *DCLR* and *DCRL* programs are the same, since they are singly recursive, and their *compose* relation is *append*, which is associative. For the binary tree *flat* relations and for *quicksort*, the *DCRL* programs are much better than the

DCLR programs, because of properties of relations like *append* (which is the *compose* relation in all these programs), which are the main reason for achieving the optimisations of the *DCRL* programs for the relations above. Hence, if the input programs for the binary tree *flat* relations and for the *quicksort* problem to the duality schema are instances of the *DCLR* schema pattern, then a duality transformation will be performed, resulting in *DCRL* programs for these relations, since both the applicability and the optimisability conditions are satisfied by these programs. If the *DCRL* programs for the relations above are input to the duality schema, then the duality transformation will *not* be performed, since the optimisability conditions are not satisfied by *append*, which is the *compose* relation of the *DCRL* programs. Of course, there may exist some other relations where the duality transformation of their *DCRL* programs into the *DCLR* programs will provide an efficiency gain. Unfortunately, we could not find a meaningful relation of this category.

The next step in evaluating the transformation schemas is to compare the generalised programs of these example relations. If we look at Table 1, the most obvious observation is that the *DGRL* programs for all these relations are very efficient programs. However, tupling generalisation seems to be the second best as a generalisation choice, and it must even be the first choice for relations like infix *flat*, where the composition place of the head in the result parameter is in the middle, and where the *minimal* and *nonMinimal* checks can be performed in minimum time. Although a similar situation occurs for *quicksort*, its TG program is not quite as efficient as its *DGRL* program. This is mainly because of *partition*, which is the *decompose* relation of *quicksort*, being a costly operation, although we eliminated most of the *partition* calls by putting extra minimality checks into the TG template. Since *append*, which is the *compose* relation in all the programs, cannot be eliminated in the resulting *DGLR* programs, the *DGLR* programs for these relations have the worst timing results. The reason for their bad performance is that the percentages of the total running times of the *DGLR* programs used by *append* are much higher than the percentages of the total running times of the *DCLR* and *DCRL* programs used by *append* for these relations. The reason for the increase in the percentages is that the length of the accumulator, which is the input parameter to *append* in the *DGLR* programs, is larger than the length of the input parameter of *append* in the *DCLR* and *DCRL* programs, since the partial result has to be repeatedly input to the *compose* relation in descending generalisation.

A transformation should be performed only if it really results in a program that is more efficient than the input program. So, for instance, the descending generalisation of the input *DCLR* program for infix *flat* resulting in the *DGLR* program must not be done, even though the applicability conditions are satisfied. This is the main reason for the existence of the optimisability conditions in the schemas.

In some of the cases, using generalisation schemas to transform input programs that are already generalised programs into DC programs can produce an efficiency gain. For example, if the *DGLR* program for any of the *flat* relations is the input program to descending generalisation (namely DG_1 or DG_4), then a de-generalisation will be performed resulting in the $DCLR$ (or $DCRL$) program, which is more efficient than the input $DGLR$ program. However, with the current optimisability conditions, if the input program for any of the relations above to generalisation is a $DGRL$ program, then the generalisation schemas are still applied in the reverse direction, resulting in a $DCRL$ program, which means that the de-generalisation will result in a program that is less efficient than the input program. This makes us think of even more accurate ways of defining the optimis*ability* conditions, namely as actual optimis*ation* conditions, such that the transformation will always result in a better program than the input program. However, more performance analyses and complexity analyses are needed to derive such conditions.

7 A Prototype Transformation System

TRANSYS is a prototypical implementation of the schema-based program transformation approach summarised in this paper. TRANSYS is a fully automatic program transformation system and was developed to be integrated with a schema-guided program development environment. Therefore, the input program to TRANSYS is assumed to be developed by a synthesiser using the database of schema patterns known to TRANSYS. The schema pattern of which the input program is an instance is thus a priori known, and so are the renamings of the open relation symbols, the particularisations of the schema variables such as t and p, as well as the "closing" programs defining these open relations of the template. In other words, no matching between the input program and the templates of the transformation schemas has to be performed, unlike in [6,25]. Given an input program, TRANSYS outputs (what it believes to be) the best programs that are more efficient than the input program: this is done by collecting the leaves of the tree rooted in that input program and where child nodes are developed when both the applicability and the optimisability conditions of a transformation schema hold. All the transformation schemas and the schema patterns, which are the input (or output) schema patterns of these transformation schemas, given in [5] (i.e., a superset of the schemas given in this paper), are available in the database of the system.

TRANSYS has been developed in SICStus Prolog 3. Since TRANSYS is a prototype system, for some parts of the system, instead of implementing them ourselves, we reused and integrated other systems:

– For verifying the applicability conditions and some of the optimisability conditions, PTTP is integrated into the system. The *Prolog Technology Theorem Prover* (PTTP) was developed by M. Stickel (for a detailed explanation of PTTP, the reader can refer to [24]). PTTP is an implementation of the model elimination theorem proving procedure for the full first-order predicate calculus. TRANSYS uses the version of PTTP that is written in Prolog and that compiles clauses into Prolog.

– For verifying the other optimisability conditions, and for applying these optimisations to the output programs of the transformation schemas, we integrated Mixtus 0.3.6. Mixtus was developed by D. Sahlin (for a detailed explanation of Mixtus, the reader can refer to [19]). Mixtus is an automatic partial evaluator for full Prolog. Given a Prolog program and a query, it will produce a new program specialised for all instances of that query. The partial evaluator is guaranteed to terminate for all input programs and queries.

For a detailed explanation of the TransSys system, the reader is invited to consult [5].

8 Conclusions and Future Work

This paper results from the research that began by investigating the suggestions in [11]. The contributions of this research are:

– pre-compilation of more general generalisation schemas (tupling and descending) than those in [11], which were restricted to sub-families of divide-and-conquer programs;
– discovery of the duality schemas;
– discovery of the optimisability conditions;
– validation of the correctness of the transformation schemas, based on the notions of correctness of a program, steadfastness of a program in a set of specifications, and equivalence of two programs (the correctness proofs of the transformation schemas given in this paper and in [5] can be found in [4]; another approach to validation of transformation schemas can be found in [18]);
– development of a prototype transformation system;
– validation of the effectiveness of the transformation schemas by performance tests.

This research opens future work directions, such as:

– extension to normal programs and open frameworks;
– consideration of other program schemas (or schema patterns);
– extension of the schema pattern language so as to express even more general program families;
– representation of the loop merging strategy as a transformation schema;
– search for other transformation schemas;
– identification of optimis*ation* conditions that *always* ensure improved performance (or complexity) of the output program wrt the input program;
– validation of the effectiveness of the transformation schemas by automated complexity analysis (using GAIA [7] and/or CASLOG [8]).

Acknowledgments. We wish to thank the anonymous reviewers of the previous versions of this paper as well as the participants of the LOPSTR'97 workshop for their valuable comments and suggestions, especially Yves Deville (UC Louvain, Belgium). We also gratefully acknowledge the feedback of the students of the second author's Automated Software Engineering course at Bilkent, especially Necip Fazıl Ayan, Brahim Hnich, Ayşe Pınar Saygın, Tuba Yavuz, and Cemal Yılmaz.

References

1. T. Batu. *Schema-Guided Transformations of Logic Algorithms.* Senior Project Report, Bilkent University, Department of Computer Science, 1996.
2. R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems* 6(4):487–504, 1984.
3. R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.
4. H. Büyükyıldız and P. Flener. *Correctness Proofs of Transformation Schemas.* Technical Report BU-CEIS-9713. Bilkent University, Department of Computer Science, 1997.
5. H. Büyükyıldız. *Schema-based Logic Program Transformation.* M.Sc. Thesis, Technical Report BU-CEIS-9714. Bilkent University, Department of Computer Science, 1997.
6. E. Chasseur and Y. Deville. Logic program schemas, semi-unification, and constraints. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97* (this volume).
7. A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of Prolog programs. In: J. Gallagher (ed), *Proc. of LOPSTR'96*, pp. 38–57. LNCS 1207. Springer-Verlag, 1997.
8. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM TOPLAS* 15(5):826–875, 1993.
9. Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.
10. Y. Deville and J. Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In: E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'89*, pp. 409–425. The MIT Press, 1989.
11. P. Flener and Y. Deville. Logic program transformation through generalization schemata. In: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 171–173. LNCS 1048. Springer-Verlag, 1996.
12. P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97* (this volume).
13. N.E. Fuchs and M.P.J. Fromherz. Schema-based transformation of logic programs. In: T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'91*, pp. 111–125. Springer Verlag, 1992.
14. T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In: L. Sterling (ed), *Proc. of ICLP'95*, pp. 467–481. The MIT Press, 1995.
15. T.S. Gegg-Harrison. Extensible logic program schemata. In: J. Gallagher (ed), *Proc. of LOPSTR'96*, pp. 256–274. LNCS 1207. Springer-Verlag, 1997.
16. A. Hamfelt and J. Fischer Nilsson. Declarative logic programming with primitive recursion relations on lists. In: L. Sterling (ed), *Proc of JICSLP'96.* The MIT Press.

17. A. Pettorossi and M. Proietti. Transformation of logic programs: foundations and techniques. *Journal of Logic Programming* 19(20):261–320, 1994.
18. J. Richardson and N.E. Fuchs. Development of correct transformation schemata for Prolog programs. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97* (this volume).
19. D. Sahlin. *An Automatic Partial Evaluator of Full Prolog*. Ph.D. Thesis, Swedish Institute of Computer Science, 1991.
20. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.
21. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering* 16(9):1024–1043, 1990.
22. Z. Somogyi, F. Henderson, and T. Conway. Mercury: An efficient purely declarative logic programming language. In: *Proc. of the Australian Computer Science Conference*, pp. 499–512, 1995.
23. L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In: J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 127–140, John Wiley, 1993.
24. M.E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. *Theoretical Computer Science* 104:109–128, 1992.
25. W.W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 174–188. LNCS 1048. Springer-Verlag, 1996.