Logic Program Transformation through Generalization Schemata

Pierre Flener

Department of Computer Engineering and Information Science
Faculty of Engineering, Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr Voice: +90/312/266-4000 ext.1450

Yves Deville

Department of Computing Science and Engineering
Université Catholique de Louvain, B–1348 Louvain-la-Neuve, Belgium
Email: yde@info.ucl.ac.be Voice: +32/10/47-2067

Abstract

In program synthesis, program transformation can be done on the fly, based on information generated and exploited during the program construction process. For instance, some logic program generalization techniques can be pre-compiled at the logic program schema level, yielding transformation schemata that give rise to elimination of the eureka discovery, full automation of the transformation process itself, and even the prediction whether and which optimizations will be achieved.

1 Introduction

Programs can be classified according to their construction methodologies, such as divide-and-conquer, generate-and-test, top-down decomposition, global search, and so on, or any composition thereof. Informally, a *program schema* is a template program with a fixed control and data flow, but without specific indications about the actual parameters or the actual computations, except that they must satisfy certain constraints. A program schema thus abstracts a whole family of particular programs that can be obtained by instantiating its place-holders to particular parameters or computations, using the specification, the program synthesized so far, and the constraints of the schema. It is therefore interesting to guide program construction by a schema that captures the essence of some methodology. This reflects the conjecture that experienced programmers actually instantiate schemata when programming, which schemata are summaries of their past programming experience. For a more complete treatise on this subject, please refer to [3] [4].

Moreover, in contrast to traditional programming methodologies where program transformation sequentially follows program construction and is merely based on the constructed program, these two phases can actually be interleaved in (semi-)automated program synthesis, based on information generated and exploited during the program construction process.

In this introductory section, we introduce a schema for divide-and-conquer logic programs (Section 1.1), and argue for schema-guided transformation (Section 1.2). Finally, we introduce the notion of problem generalization (Section 1.3).

1.1 A Divide-and-Conquer Logic Program Schema

In this sub-section, for the purpose of illustration only, we focus on the divide-and-conquer construction methodology, and we restrict ourselves, for pedagogical purposes only, to binary predicates. A *divide-and-conquer program* for a binary predicate R over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is (usually) easily found by directly solving the problem. Otherwise, that is if X is non-minimal, decompose (or: divide) X into a vector $\boldsymbol{HX}$ of h heads of X and a vector $\boldsymbol{TX}$ of t tails of X, the tails being of the same type as X, as well as smaller than X according to some well-founded relation. The t tails $\boldsymbol{TX}$ recursively yield t tails $\boldsymbol{TY}$ of Y (this is the conquer step). The h heads $\boldsymbol{HX}$ are processed into a vector $\boldsymbol{HY}$ of h' heads of Y. Finally, Y is composed (or: combined) from its heads $\boldsymbol{HY}$ and tails $\boldsymbol{TY}$. Suppose m subcases with different processing and composition operators emerge: one discriminates between them according to the values of $\boldsymbol{HX}$, $\boldsymbol{TX}$, and Y. The m+1 clauses of logic programs synthesized by this divide-and-conquer methodology are thus covered by the second-order clause schemata of Schema 1, where $\boldsymbol{R(TX,TY)}$ stands for the conjunction of the $R(TX_j,TY_j)$, for $1 \le j \le t$:

```
R(X,Y) ←
   Minimal(X),
   Solve(X,Y)
R(X,Y) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   Discriminate_k(HX,TX,Y),
   R(TX,TY),
   Process_k(HX,HY),
   Compose_k(HY,TY,Y)
```

Schema 1: Divide-and-conquer schema

Note that, upon instantiation, the second clause schema yields m different clauses, for $1 \le k \le m$. The constraints to be verified by first-order instances of this schema are listed in [4]. The most important one is that there must exist a well-founded relation "$<$" over the domain of the induction parameter, such that the instantiation of *Decompose* guarantees that TX_j "$<$" X, for every $1 \le j \le t$.

Example 1: Let *sort*(*L*,*S*) hold iff integer-list S is a non-decreasingly ordered permutation of integer-list L The following insertion-sort program:

```
sort([],[])
sort([HL|TL],S) ←
   sort(TL,TS),
   insert(HL,TS,S)
```

is a rewriting of the program obtained by applying the substitution {*Minimal*/*isEmptyList*, *Solve*/=, *NonMinimal*/*isNonEmpty*, *Decompose*/*headTail*, $Discriminate_1$/*true*, $Process_1$/=, $Compose_1$/*insert*, m/1, h/1, h'/1, t/1} to Schema 1, where the used primitives are defined as follows:

```
isEmptyList([])        isNonEmpty([_|_])        headTail([H|T],H,T)
```

and *insert*(*I*,*L*,*R*) holds iff list R is non-decreasing integer-list L with integer I inserted. Note that a logic program for *insert*/3 would not be an instance of Schema 1: covering it would require a generalization to n-ary predicates, the handling of *non*-recursive non-minimal clauses, and the handling of auxiliary parameters (such as I), which can't be induction parameters. Such extensions are discussed in [3] [4]. ♦

1.2 Schema-Guided Transformation

Regarding program transformation, it would be interesting to pre-compile transformation techniques at the schema-level: a *transformation schema* is a pair $\langle T_1, T_2 \rangle$ of program schemata, such that T_2 is a transformation of T_1, according to some transformation technique. Transformation at the program-level then reduces to (*i*) selecting a transformation schema $\langle T_1, T_2 \rangle$ such that T_1 covers the given program under some substitution σ, and (*ii*) applying σ to T_2. This is similar to the work of Fuchs and Fromherz [6], except that they have a preliminary abstraction phase where the covering schema of the given program is discovered,

whereas we here assume that this program was synthesized in a schema-guided fashion, so that the covering schema is already known. Moreover, their work is much more abstract than ours in the sense that they don't say much about the contents of the schemata, whereas we here assume that the given program is covered by a divide-and-conquer schema. Waldau [8] shows how to formally validate such transformation schemata.

1.3 Problem Generalization

When synthesizing a divide-and-conquer program, it sometimes becomes "difficult," if not impossible, to process-compose *Y* from the ***HX*** and ***TY***. Besides changing the induction parameter, or the well-founded relation over its domain (and hence the instances of *Minimal*, *NonMinimal*, and *Decompose*), or the guiding schema/methodology, one can also generalize the initial specification and then synthesize a recursive program from the generalized specification as well as a non-recursive program for the initial problem as a particular case of the generalized one. Paradoxically, the new synthesis then becomes "easier," if not possible in the first place. As an additional and beneficial side-effect, programs for generalized problems are often more efficient, because they feature (semi-)tail recursion (which is transformed by optimizing interpreters into iteration) and/or because the generalization provoked a complexity reduction by loop merging.

For a detailed overview of problem generalization techniques, the reader is invited to consult [1] or [2]. We here summarize Deville's presentation and borrow some of his examples, but adapt and extend where appropriate. Basically, there are two generalization approaches:

- In *structural generalization*, one generalizes the structure (type) of some parameter. For instance, an integer parameter could be generalized into a integer-list parameter, and the intended relation must then be generalized accordingly. This is called *tupling generalization*, and we shall restrict the discussion of structural generalization to it.
- In *computational generalization*, one generalizes a state of computation in terms of "what has already been done" and "what remains to be done." If information about what has already been done is not needed, then it is called *descending generalization*, otherwise it is *ascending generalization*.

The remainder of this paper is organized as follows. In Sections 2 and 3 respectively, we show how tupling generalization and descending generalization can be applied for the purposes of pre-compiled automatic program transformation. (It is yet unclear to us how to do this with ascending generalization.) This yields transformation schemata called *generalization schemata*. Finally, in Section 4, we conclude.

2 Program Transformation by Structural Generalization

In this section, we first briefly illustrate tupling generalization on an example, and then examine its potential for automatic logic program transformation.

Example 2: Let the constant *void* represent the empty binary tree, and the compound term *btree*(*L*,*E*,*R*) represent a binary tree of root *E*, left subtree *L*, and right subtree *R*. Let *flat*(*B*,*F*) hold iff list *F* contains the elements of binary tree *B* as they are visited by a prefix traversal of *B*. We also say that *F* is the prefix representation of *B*. A corresponding "naive" divide-and-conquer logic program could be:

```
flat(void,[])
flat(btree(L,E,R),F) ←
   flat(L,U), flat(R,V),
   H=[E],
   append(U,V,I), append(H,I,F)
```

where *append*(*A*,*B*,*C*) holds iff list *C* is the concatenation of list *B* to the end of list *A*. If *n* is the number of elements in tree *B*, then this logic program has an $O(n^2)$ time complexity (as opposed to the linear complexity one might expect), because composition is done through *append*/3, whose time complexity is linear in the number of elements in its first parameter (and not constant, as one might hope). This is all an inevitable consequence of the definition of lists. Worse, if *h* is the height of *B*, then this logic program builds a stack of *h* pairs of recursive calls, and creates $2 \cdot n$ intermediate data structures, so it also has a very bad space complexity. In the (*in*,*out*) mode, the calls to the composition operator *append*/3 cannot be moved in front of any of the recursive calls (well, at least not without significantly further degrading the time/space efficiency), so the *flat*/2 logic program is not (semi-)tail-recursive (assuming a left-to-right computation rule).

Let's now perform a tupling generalization of the initial specification: let *flats*(*Bs*,*F*) hold iff list *F* is the concatenation of the prefix representations of the elements of binary tree list *Bs*. Expressing the initial problem as a particular case of the generalized one yields:

```
flat(B,F) ←
   flats([B],F)
```

and a logic program for the generalized problem is:

```
flats([],[])
flats([void|Bs],F) ←
   flats(Bs,F)
flats([btree(L,E,R)|Bs],[E|TF]) ←
   flats([L,R|Bs],TF)
```

Note that the calls to *append*/3 have disappeared: the *append*/3 loops have been merged into the *flat*/2 loop. So the conjunction of the last four clauses yields the ideal $O(n)$ time complexity. This logic program builds a stack of $2 \cdot n+1$ recursive calls, and it creates as many intermediate data structures; fortunately, the logic program for *flats*/2 can be made tail-recursive in the mode (*in*,*out*), as the last two clauses are exclusive. ♦

The *eureka* needed to adequately generalize the initial specification comes directly from the composition operator of the initial program [5]. For instance, *flats*/2 was specified in terms of a *concatenation* precisely because the composition operator of *flat*/2 was *append*/3. Generally speaking now, suppose the specification of the initial problem is (where $\mathcal{R}$ is the intended relation, and $\mathcal{T}_1$, $\mathcal{T}_2$ are the types of its parameters):

$R(X,Y) \Leftrightarrow \mathcal{R}[X,Y]$,
where $X \in \mathcal{T}_1$ and $Y \in \mathcal{T}_2$.

and that its initially synthesized divide-and-conquer program is covered by the following specialization (for simplicity of presentation only) of Schema 1:

```
R(X,Y) ←
   Minimal(X),
   Solve(X,Y)
R(X,Y) ←
   NonMinimal(X),
   Decompose(X,HX,TX1,TX2),
   R(TX1,TY1), R(TX2,TY2),
   Process(HX,HY),
   Compose(TY1,TY2,I), Compose(HY,I,Y)
```

Schema 2: Divide-and-conquer schema (where h=h'=m=1 and t=2)

(note that this may require the rewriting of $Process_1$ so that it type-transforms *HX* into a term *HY* of type $\mathcal{T}_2$, and/or the rewriting of $Compose_1$ so that it is a conjunction of calls to some *Compose*/3, which must be associative and must have some left/right-identity element *e*).
The formal specification of the tupling-generalized problem then is:

$Rs(Xs,Y) \Leftrightarrow$
$(\, Xs = [] \wedge Y = \varepsilon \,) \vee (\, Xs = [X_1,X_2,\ldots,X_n] \wedge \boldsymbol{R}(X_i,Y_i) \wedge I_1 = Y_1 \wedge \boldsymbol{Compose}(I_{i-1},Y_i,I_i) \wedge Y = I_n \,)$,
where $X \in$ *list of* $\mathcal{T}_1$ and $Y \in \mathcal{T}_2$.

and the new logic program is:

```
R(X,Y) ←
   Rs([X],Y)                                          (1)
```

where:

```
Rs(Xs,Y) ←
   Xs=[],                    % Minimal-Rs
   Y=e                       % Solve-Rs
Rs(Xs,Y) ←
   Xs=[_|_],                 % NonMinimal-Rs
   Xs=[X|TXs],               % Decompose1-Rs
```

```
   Minimal(X),                 % Discriminate_1-Rs
   Rs(TXs,TY),
   Solve(X,HY),                % Process_1-Rs
   Compose(HY,TY,Y)            % Compose_1-Rs
Rs(Xs,Y) ←
   Xs=[_|_],                   % NonMinimal-Rs
   Xs=[X|TXs],                 % Decompose_2-Rs (part 1)
   NonMinimal(X),              % Discriminate_2-Rs
   Decompose(X,HX,TX_1,TX_2),  % Decompose_2-Rs (part 2)
   Rs([TX_1,TX_2|TXs],TY),
   Process(HX,HY),             % Process_2-Rs
   Compose(HY,TY,Y)            % Compose_2-Rs
```

Schema 3: Tupling generalization schema, expressed using the divide-and-conquer operators

This is easily proved by unfolding the body of (1) with Schema 3, and transforming into Schema 2 again [5]. The annotations clearly show that logic programs for tupling-generalized problems are covered by a slight generalization of the divide-and-conquer schema (Schema 1). Structural generalization is thus very easy if a (naive) program is already given, as this technique is very suitable for mechanical transformation: all operators of the generalized program are operators of the initial program.

Moreover, if *Solve*/2 converts *X* into a constant "size" *Y*, then the conjunction `Solve(X,HY), Compose(HY,TY,Y)` of the second clause can be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. Often, this partial evaluation even results in an equality atom for *Y*, which can then be forward-compiled (into the head of the second clause). The second and third clauses being mutually exclusive (by virtue of a constraint that *Minimal*/1 and *NonMinimal*/1 must be complementary over the domain of the induction parameter *X*), the recursive call `Rs(TXs,TY)` in the second clause can then be made iterative (by, *e.g.*, placing a cut after the call to *Minimal*/1). For instance, the prefix representation of the empty tree is the empty list (of size 0), so partial evaluation gives $F = TF$, which can indeed be compiled into the head of the second clause.

Finally, if *Process*/2 converts *HX* into a constant "size" *HY*, then the conjunction `Process(HX,HY), Compose(HY,TY,Y)` of the third clause can also be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. Often, this partial evaluation even results in an equality atom for *Y*, which can then be forward-compiled (into the head of the third clause), so that the recursive call `Rs([TX_1,TX_2|TXs],TY)` in the third clause also becomes iterative. For instance, the list $[E]$ obtained through processing of the root *E* being of size/length 1, partial evaluation gives $F = [E \mid TF]$, which can indeed be compiled into the head of the third clause.

If the last two conditions simultaneously hold (which is not unusual), then *Compose*/3 effectively disappears altogether. So, given a divide-and-conquer program, a mere inspection of the properties of its solving, processing, and composition operators allows us to detect whether tupling generalization is possible, and even to which optimizations it would lead. Even better, the *eureka* discovery is compiled away, and the transformation can be completely automated. The pair ⟨Schema 2, Schema 3⟩ thus constitutes a first generalization schema.

This presentation generalizes the one of [1] and [2], where $\mathcal{T}_2$ must be *list* (which is a useless restriction, because it is unrelated to the nature of tupling generalization), and where *Compose*/3 must be *append*/3 and *e* must be [] (which hides the fact that the *eureka* is always based on *Compose*/3, which can be different from *append*/3). Also, Schema 3 was not discovered there, hence the possibility of full pre-compilation went undetected there.

Moreover, in [1, p.139], the potential existence of structural cases other (or even more numerous) than [] and [_|_], with the first element either minimal or not, is identified. For instance, if *flat*/2 was to compute the infix representation of a binary tree, then the cases [], $[void \mid Bs]$, $[btree(void,E,R) \mid Bs]$, and $[btree(L,E,R) \mid Bs] \wedge L \neq void$ would be needed for *flats*/2. This might seem disturbing at first sight, as it seems to prevent full automation. But this simply follows from the fact that the program for *flat*/2 would *not* be covered by Schema 2, but rather by a slight variant thereof, namely where the composition conjunction `Compose(TY_1,TY_2,I), Compose(H,I,Y)` is replaced by `Compose(HY,TY_2,I), Compose(TY_1,I,Y)`. Pre-compilation of the transformation yields the corresponding variant of Schema 3.

One can thus pre-compile more generalization schemata, one for each combination of values of $\langle h, h', t, m \rangle$ and each ordering of composition of Y from HY and the $\boldsymbol{TY}$ [5].

3 Program Transformation by Descending Generalization

In this section, we first briefly illustrate descending generalization on the classical list reversal example, and then examine its potential for automatic logic program transformation.

Example 3: Let *reverse*(*L*,*R*) hold iff list *R* is the reverse of list *L*. A corresponding "naive" divide-and-conquer logic program could be:

```
reverse([],[])
reverse([HL|TL],R) ←
   reverse(TL,TR),
   HR=[HL],
   append(TR,HR,R)
```

If n is the number of elements in list L, then this logic program has an $O(n^2)$ time complexity (as opposed to the linear complexity one might expect), because composition is done through *append*/3, whose time complexity is linear in the number of elements in its first parameter. Worse, this logic program builds a stack of n recursive calls, and creates n intermediate data structures, so it also has a very bad space complexity. In the (*in*,*out*) mode, the call to the composition operator *append*/3 cannot be moved in front of the recursive call (well, at least not without significantly further degrading the time/space efficiency), so the *reverse*/2 logic program is not tail-recursive (assuming a left-to-right computation rule).

Let's now perform a descending generalization of the initial specification: let *reverseDesc*(*L*,*R*,*A*) hold iff list *R* is the concatenation of list *A* to the end of the reverse of list *L*. Expressing the initial problem as a particular case of the generalized one yields:

```
reverse(L,R) ←
   reverseDesc(L,R,[])
```

and a logic program for the generalized problem is:

```
reverseDesc([],R,R)
reverseDesc([HL|TL],R,A) ←
   reverseDesc(TL,R,[HL|A])
```

Note that the call to *append*/3 has disappeared: the *append*/3 loop has been merged into the *reverse*/2 loop. So the conjunction of the last three clauses yields the ideal $O(n)$ time complexity. This logic program also builds a stack of n recursive calls, but it creates no intermediate data structures; fortunately, the logic program for *reverseDesc*/3 is even tail-recursive in the mode (*in*,*out*,*in*). ♦

In its simplest incarnation, descending generalization thus introduces an accumulator parameter, which is progressively extended to the final result. The pair of parameters R and A can also be seen as representing the difference-list $R \setminus A$, which itself represents the difference between lists R and A, where A is a suffix of R. But descending generalization yields something more general than transformation to difference-list manipulation, because it is by no means restricted to creating difference-lists only: any form of difference-structures can be created. Another example would be difference-integer $I \setminus J$, which could represent $I–J$ or $max(I,J)$ or whatever.

It can again be shown that the *eureka* needed to adequately generalize the initial specification comes directly from the composition operator of the initial program [5]. For instance, *reverseDesc*/3 was specified in terms of a *concatenation* precisely because the composition operator of *reverse*/2 was *append*/3. Its formal specification, namely:

$$reverseDesc(TL,R,HR) \Leftrightarrow \exists TR \;\; reverse(TL,TR) \wedge append(TR,HR,R)$$

has a right-hand side built of two atoms extracted from the logic program for *reverse*/2. Generally speaking now, suppose the initially synthesized divide-and-conquer program for the initial problem is covered by the following specialization (for simplicity of presentation only) of Schema 1:

```
R(X,Y) ←
   Minimal(X),
   Solve(X,Y)
R(X,Y) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   R(TX,TY),
   Process(HX,HY),
   Compose(HY,TY,Y)
```

Schema 4: Divide-and-conquer schema (where h=t=m=1)

The *eureka* can then be mechanically found [1] [7] by searching in the program for $R/2$ for a (not necessarily consecutive) sub-formula of the form `R(X,S), Compose(A,S,Y)`, so that the following formal specification for *R-desc*/3 can be postulated:

$$\textit{R-desc}(X,Y,A) \Leftrightarrow \exists S \;\; R(X,S) \wedge \textit{Compose}(A,S,Y)$$

The key principle here is that both parts of the sub-formula share some variable S. The same principle can be employed for other loop mergers, but we are here only interested in descending generalization. This is of course not a new result, but the following further development has some new ideas. This search is easy if the program for $R/2$ was constructed in the first place so as to be an instance of Schema 4. Note that it is thus crucial that the *Process*/2 and *Compose*/3 operators are not merged (yet).

Now, if the initial problem exhibits a functional dependency from induction parameter X to parameter Y, and if *Compose*/3 is associative with left-identity element e, then the new logic program is:

```
R(X,Y) ←
   R-desc(X,Y,e)                                          (2)
```

where:

```
R-desc(X,Y,A) ←
   Minimal(X),
   ( Solve(X,S), Compose(A,S,Y) )
R-desc(X,Y,A) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   ( Process(HX,HI), Compose(A,HI,NewA) ),
   R-desc(TX,Y,NewA)
```

Schema 5: Descending generalization schema, expressed using the divide-and-conquer operators

This is easily proved by unfolding the body of (2) with Schema 5, and transforming into Schema 4 again [5]. Logic programs for descendingly generalized problems are obviously *not* covered by a divide-and-conquer schema, because the accumulator is extended for recursive calls rather than reduced. The corresponding tail-recursive schema is actually as follows:

```
R-desc(X,Y,A) ←
   Minimal(X),
   ExtendMin(X,A,Y)
R-desc(X,Y,A) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   ExtendNonMin(HX,A,NewA),
   R-desc(TX,Y,NewA)
```

Schema 6: Descending generalization schema

Descending generalization is thus very easy if a (naive) program is already given, as this technique is very suitable for mechanical transformation: all operators of the generalized program are operators of the initial program.

Moreover, if the intended relation behind $R/2$ maps the minimal form of parameter X into e, and if e is also a right-identity element of *Compose*/3, then the conjunction `Solve(X,S), Compose(A,S,Y)`

of the first clause can be simplified into `Y=A`. For instance, the reverse of the empty list is the empty list, which is indeed the right-identity element of *append*/3.

Finally, if *Process*/2 converts *HX* into a constant "size" *HY*, then the atom `Compose(A,HI,NewA)` in the second clause can be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. For instance, the processing operator of *reverse*/2 maps the first element *HL* of a non-empty list into the singleton list [*HL*], so the atom `append([HL],A,NewA)` can indeed be partially evaluated, namely into `NewA=[HL|A]`. Further transformations yield the *reverseDesc*/3 program above. However, for *minList*(*L*,*M*), which holds iff integer *M* is the minimum element of integer-list *L*, the processing operator, namely =/2, maps the head of the list to itself, that is to an arbitrary integer, so that call to the composition operator, namely *min*/3, cannot be partially evaluated away. Generally speaking, if the elements of *X* are of the same type as *Y*, then that call to *Compose*/3 cannot be partially evaluated away, because the elements of *X* are usually not of a constant size, and hence are not processed into constant size *HY*.

If the last two conditions simultaneously hold (which is not unusual), then *Compose*/3 effectively disappears altogether. So, given a divide-and-conquer program, a mere inspection of the properties of its processing and composition operators allows us to detect whether descending generalization is possible and even to which optimizations it would lead. Even better, the *eureka* discovery is compiled away, and the transformation can be completely automated. The pair ⟨Schema 4, Schema 5⟩ thus constitutes another generalization schema.

This presentation generalizes the one of [1] and [2], where Schema 5 was not discovered, and hence the possibility of full pre-compilation undetected. This also generalizes the presentation of [8], where a simpler schema than Schema 4 was used. Further generalizations can be found in [5].

4 Conclusion

We have shown that some logic program generalization techniques can be pre-compiled at the program schema level, so that the corresponding transformation can be fully automated, including the elimination of the *eureka* discovery and the detection whether and which optimizations will result from the transformation. This subsequently allows logic program synthesis to perform program transformations during the program construction process, based on information that the latter generated and exploited.

References

[1] Yves Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.

[2] Yves Deville and Jean Burnay. Generalization and program schemata. In E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'*89, pp. 409–425. The MIT Press, 1989.

[3] Pierre Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publ., 1995.

[4] Pierre Flener. *Logic Program Schemata: Synthesis and Analysis*. Technical Report BU-CEIS-9502, Bilkent University, Ankara (Turkey), 1995. Submitted for publication.

[5] Pierre Flener and Yves Deville. *Logic Program Transformation through Generalization Schemata*. Technical Report BU-CEIS-95xx, Bilkent University, Ankara (Turkey), 1995. In preparation.

[6] Norbert E. Fuchs and Markus P.J. Fromherz. Schema-based transformations of logic programs. In T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'*91, pp. 111–125. Springer-Verlag, 1992.

[7] Maurizio Proietti and Alberto Pettorossi. Synthesis of eureka predicates for developing logic programs. In N. Jones (ed), *Proc. of ESOP'*90. LNCS 432:306–325. Springer-Verlag, 1990.

[8] Mattias Waldau. Formal validation of transformation schemata. In T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'*91, pp. 97–110. Springer-Verlag, 1992.