Developing Correct and Efficient Logic Programs by Transformation *

Alberto Pettorossi
Department of Informatics, University of Roma II
Via della Ricerca Scientifica, 00133 Roma, Italy. `adp@iasi.rm.cnr.it`

Maurizio Proietti
IASI-CNR
Viale Manzoni 30, 00185 Roma, Italy. `proietti@iasi.rm.cnr.it`

1 Introduction

The complex process of deriving programs from specifications is often divided into the following three steps: i) the derivation of formal specifications from the informal ones, ii) the validation of the formal specifications, and iii) the derivation of executable programs from the formal specifications.

Each step of this derivation process can be supported by the use of elegant and well-understood notions of mathematical logic. In particular, from informal specifications given as sentences in a restricted form of the natural language, one can derive formal specifications as formulas of a first order logical theory [20]. One may then validate the derived formal specifications by checking whether or not they are consistent and satisfy some suitable properties. Finally, as we will illustrate in this paper, from formal specifications one may obtain executable, efficient programs by using techniques for transforming logic programs. This is, indeed, one of the reasons that makes logic programming very attractive for program construction. During this final step from specifications to programs, in order to improve efficiency one may want to use program transformation for avoiding multiple visits of data structures, or replacing complex forms of recursion by tail recursion, or reducing nondeterminism of procedures.

This paper is structured as follows. In Section 2 we present the rule-based approach to program transformation and its use for the derivation and synthesis of logic programs from specifications. In Section 3 we consider the schema-based transformation technique for the development of efficient programs. In Section 4 we consider the partial evaluation technique and its use for the specialization of logic programs when the input data are partially known at compile time. In the final section we discuss some of the achievements and challanges of program transformation as a tool for logic-based software engineering.

For simplicity reasons in this paper we will only consider definite logic programs, although most of the techniques we will describe can be applied also in the case of general logic programs. We refer to [35, 41] for all notions concerning logic programming and logic program transformation which are not explicitly presented here.

*In: *Knowledge Engineering Review*, Vol. 11, No. 4, December 1996, pp. 347–360.
This work has been partially supported by the European Community under the Human Capital and Mobility Project "Logic Program Synthesis and Transformation", the Project MURST 40%, and the Italian National Research Council Contract no. 95.000212.CT07.

2 Rule-based Program Transformation

The rule-based approach to logic program transformation, also called 'rules + strategies' approach or unfold/fold approach, has been introduced by Tamaki and Sato in [59]. They basically follow the ideas of Burstall and Darlington [11] where rule-based program transformation has been first described in the case of functional programs. In this approach a given program is transformed, maybe in several steps, into a new, more efficient program by applying suitable transformation rules according to some given strategies.

Now we present the program transformation rules we consider in this paper. We assume that from an initial program P_0 we have obtained by program transformation the sequence $P_0, \ldots, P_k$ of programs. The next program in the sequence, call it P_{k+1}, is obtained by an application of one of the following rules, where by $hd(C)$ and $bd(C)$ we denote the head and the body, respectively, of a clause C.

Definition rule. Program P_{k+1} is obtained by adding to program P_k a new clause of the form $p(X_1, \ldots, X_m) \leftarrow B$, where B is a non-empty sequence of atoms and $X_1, \ldots, X_m$ are distinct variables occurring in B. The predicate symbol p is a new symbol not occurring in $P_0, \ldots, P_k$, and all predicates occurring in B also occur in at least one of the programs $P_0, \ldots, P_k$.
Given the sequence $P_0, \ldots, P_k$ of programs, we denote by Def_k the set of all clauses introduced by the definition rule during the construction of that sequence together with the set Def_0 which is the a subset of all clauses of P_0 of the form $p(X_1, \ldots, X_m) \leftarrow B$, where p is a predicate occurring in P_0 only once. The clauses in Def_k, for $k \geq 0$, are called *definitions*.

Unfolding Rule. Let C be a clause in P_k of the form $H \leftarrow F, A, G$, where A is an atom and F and G are (possibly empty) sequences of atoms. Let $A_1, \ldots, A_n$, with $n \geq 0$, be the clauses of program P_k, such that A is unifiable with $hd(A_1), \ldots, hd(A_n)$, with most general unifiers $\theta_1, \ldots, \theta_n$, respectively. We assume that for any i, with $1 \leq i \leq n$, C does not share any variable with A_i. Let C_i be the clause $(H \leftarrow F, bd(A_i), G)\theta_i$, for $i = 1, \ldots, n$. By *unfolding* C w.r.t. A we derive clauses $C_1, \ldots, C_n$. We get the new program P_{k+1} from program P_k by replacing C by $C_1, \ldots, C_n$. If $n = 0$ then P_{k+1} is obtained from P_k by erasing C.

Folding Rule. Let D be a clause in Def_k and C be a clause in the set $P_k - Def_k$. We assume that D does not share any variable with C. Let C be a variant of $H \leftarrow F, bd(D)\theta, G$, where F and G are (possibly empty) sequences of atoms and θ is a substitution such that (i) every variable occurring both in $bd(D)$ and $hd(D)$ is bound to a term occurring in C, and (ii) every variable occurring in $bd(D)$ and not $hd(D)$ is left unchanged. Then, by *folding* clause C w.r.t. the atom $bd(D)\theta$ using clause D we derive clause $E : H \leftarrow F, hd(D)\theta, G$ and we get the new program P_{k+1} from program P_k by replacing C by E.

Goal Replacement Rule. Let C be a clause in P_k and G_1 a goal occurring in the body of C. Suppose that for some goal G_2, in the least Herbrand model of $P_0 \cup Def_k$ we have that $\forall X_1, \ldots, X_k. (\exists Y_1, \ldots, Y_m. G_1 \leftrightarrow \exists Z_1, \ldots, Z_n. G_2)$ where: (i) $X_1, \ldots, X_k$ are all variables of G_1 also occurring outside G_1 in C, (ii) $\{Y_1, \ldots, Y_m\} = vars(G_1) - \{X_1, \ldots, X_k\}$, (iii) $\{Z_1, \ldots, Z_n\} = vars(G_2) - \{X_1, \ldots, X_k\}$, and (iv) $\{Z_1, \ldots, Z_n\} \cap vars(C) = \{\}$. Then, we get program P_{k+1} from program P_k by replacing goal G_1 by goal G_2 in the body of clause C.

Particular instances of the goal replacement rule are the *rearrangement of atoms* in bodies of clauses and the *deletion of duplicate goals*, and during program derivations we will feel free to silently use these instances of the goal replacement rule.

All transformations of definite logic programs performed by using the definition, unfolding, and folding rules are *totally correct* w.r.t. the least Herbrand model semantics, in the sense that, for each sequence $P_0, \ldots, P_k$ of programs constructed using these rules, we have that

$M(P_0 \cup Def_k) = M(P_k)$, where $M(P)$ denotes the least Herbrand model of P. However, if during the construction of $P_0, \ldots, P_k$ we also use the goal replacement rule, then in general, only partial correctness is preserved, that is, $M(P_0 \cup Def_k) \supseteq M(P_k)$. Sufficient conditions for the total correctness of transformations when goal replacement is allowed, are given in [41, 59].

Instead of the least Herbrand model semantics as a basis for the correctness of our program transformations, we could have considered other semantics, like for instance, the success set, or the finite failure set, or Prolog operational semantics. For each of these semantics one can find in the literature various sets of transformation rules which are guaranteed to be totally (or partially) correct w.r.t. those semantics (see, for instance, [41]). In particular, in order to preserve termination of Prolog programs one can give conditions which should be satisfied before rearranging atom positions within bodies of clauses [5].

Let us now look at a simple example of logic program derivation by using the rule-based program transformation approach.

Let us suppose that we want to replace all occurrences of the maximum element, say M, of a given list L of positive integers by $M-1$. This replacement can be performed by using the following program, which can be considered as a formal specification of the desired manipulation of the input list L:

1. $replmax(L, L1) \leftarrow maxl(L, M),\ repl(L, M, L1)$
2. $maxl([\,], 0) \leftarrow$
3. $maxl([H|T], M) \leftarrow maxl(T, MT),\ max(H, MT, M)$
4. $repl([\,], M, [\,]) \leftarrow$
5. $repl([H|T], M, [New|T1]) \leftarrow repl(T, M, T1),\ newelem(H, M, New)$
6. $newelem(H, M, New) \leftarrow H \neq M,\ New = H$
7. $newelem(H, M, New) \leftarrow H = M,\ New\ is\ M-1$

where the predicate $max(A, B, M)$ holds iff M is the maximum of A and B.

From this program we will derive by transformation a new, more efficient program which avoids the double visit of the list L caused by the double occurrence of L in the body of clause 1. During this transformation we will assume that $Def_0 = \{\text{clause } 1\}$ and we will use the following strategy: we repeatedly apply the unfolding and goal replacement rules starting from clause 1 with the objective of both evaluating its body in a more efficient way and also deriving clauses which can be folded using clause 1 itself. This strategy will allow us to improve via the unfolding and goal replacement steps the evaluation of the predicate *replmax*, and then we will iterate this improvement at each level of recursion by deriving, via the folding step, a recursive definition of *replmax*.

By unfolding, from clause 1 we get the following two clauses:

8. $replmax([\,], [\,]) \leftarrow$
9. $replmax([H|T], [New|T1]) \leftarrow maxl(T, MT),\ max(H, MT, M),$
   $repl(T, M, T1),\ newelem(H, M, New)$

It is impossible to fold clause 9 because in the body of clause 9 no instance of the body of clause 1 occurs. We continue unfolding clause 9 w.r.t. the atoms with predicate *maxl* and *repl*. Having done so, we get:

10. $replmax([H, K|U], [New, New1|U1]) \leftarrow maxl(U, MU),$
    $max(K, MU, MT),\ max(H, MT, M),$
    $repl(U, M, U1),\ newelem(K, M, New1),$
    $newelem(H, M, New)$

Now we may take advantage of the fact that, when clause 10 is used to evaluate a call of *replmax* with a ground list as first argument, the values of the variables H and K are known and therefore we may compute their maximum. We can do so by performing a goal replacement step based on the associativity of *max*, and we replace $maxl(U, MU)$, $max(K, MU, MT)$, $max(H, MT, M)$ by $max(H, K, Z)$, $maxl(U, MU)$, $max(Z, MU, M)$. We get:

$$11.\quad replmax([H, K|U], [New, New1|U1]) \leftarrow max(H, K, Z),\; maxl(U, MU),\; max(Z, MU, M),\; repl(U, M, U1),\; newelem(K, M, New1),\; newelem(H, M, New)$$

A folding step is still impossible. However, by looking at the history of our program derivation performed so far, we observe that in clause 11 the goal G made out of the atoms $maxl(U, MU)$, $max(Z, MU, M)$, $repl(U, M, U1)$, $newelem(K, M, New1)$ is a generalization of the body of clause 9. Indeed, the variable H which is the first argument of *max* and *newelem* in clause 9 has been generalized to the two distinct variables Z and K in clause 11. Thus, we can apply the so-called *generalization strategy* [41] by introducing the new predicate *genp* defined by the following clause whose body is the goal G itself:

$$12.\quad genp(U, Z, M, U1, K, New1) \leftarrow maxl(U, MU),\; max(Z, MU, M),\; repl(U, M, U1),\; newelem(K, M, New1)$$

The variables of the head are the ones which occur both in G and elsewhere in clause 11.

Now, starting from clause 12 and mimicking the transformation steps performed during the derivation of clause 11 from clause 9, we get:

$$13.\quad genp([\,], Z, Z, [\,], K, N) \leftarrow newelem(K, M, N)$$

$$14.\quad genp([H|T], Z, M, [New|T1], K, N) \leftarrow max(H, Z, R),\; maxl(T, MT),\; max(R, MT, M),\; repl(T, M, T1),\; newelem(K, M, N),\; newelem(H, M, New)$$

Now, a folding step is possible, and by folding clause 14 using clause 12 we get:

$$15.\quad genp([H|T], Z, M, [New|T1], K, N) \leftarrow max(H, Z, R),\; genp(T, R, M, T1, K, N),\; newelem(H, M, New)$$

We can also fold clause 9 using again clause 12 and we obtain:

$$16.\quad replmax([H|T], [New|T1]) \leftarrow genp(T, H, M, T1, H, New)$$

The final program, made out of clauses 8, 16, 13, and 15, realizes the desired list manipulation making only one visit of the input list and it is more efficient than the initial program. As already mentioned, the correctness of the final program w.r.t. the least Herbrand model semantics, does not require any proof: it is simply a consequence of the fact that the transformation rules we have used, are correct w.r.t. that semantics.

Our derivation above shows that when transforming programs we need to apply the transformation rules according to some suitable strategy. The need for a strategy is also due to the fact that we may perform useless transformations because unfolding is the inverse of folding. In particular, a suitable strategy should direct the unfolding steps which otherwise may be performed an unlimited number of times, and it should also suggest which new definitions are to be introduced and which generalizations should be made.

In the derivation of *replmax* we have seen in action the generalization strategy. It consists in discovering during the unfolding and goal replacement process a ‘similarity’ between a goal

B_C occurring in the the body of a clause C and the body B_A of a clause A which is an ancestor of C. Having discovered that similarity, the generalization strategy suggests the introduction of the definition of a new predicate via a clause whose body is the most specific generalization of both B_C and B_A. This new definition allows us both to perform a folding step of the ancestor clause A and also to derive a recursive definition of the newly introduced predicate by replaying the derivation steps which led from clause A to clause C. The generalization strategy may often improve program efficiency because the new predicate definition may exploit the interactions between the atom evaluations in its body and avoid the construction of bindings of intermediate variables or multiple visits of data structures.

The various strategies for logic program transformation may be compared w.r.t. the syntactic properties of the final programs they are able to generate, such as linear recursion, tail recursion, absence of unnecessary variables, etc. The power of the strategies may be established via completeness results based on the following definition: given a class C of programs and a set of transformation rules R, we say that a strategy S is complete w.r.t. a decidable, syntactic property φ of programs iff we can derive from a program P in C using the strategy S a new program P' which satisfies φ if it is the case that there exists a sequence of rules in R for deriving from P a program, possibly different from P', which also satisfies φ. In [46] the reader may find an example of such completeness results.

Logic program transformation using the rule-based approach is related to logic program synthesis, theorem proving, and artificial intelligence techniques. The relationship with program synthesis is based on the fact that the initial program version can be considered as a formal specification from which one has to synthesize an executable program. The relationship with theorem proving is illustrated by the fact that the generalization strategy is an instance of the generalization technique which is often used in the theorem proving field (see, for instance, [6]). By the generalization technique the proof of a given sentence is derived from that of a generalized sentence where a stronger inductive hypothesis can be used during the inductive proof. Finally, there is also a close connection between program transformation and artificial intelligence in particular in the area of the strategies for directing the application of the transformation rules for improving program efficiency.

3 Schema-based Program Transformation

During the transformational development of programs from specifications we may want to reuse previous program derivations, in particular in the case when the specification at hand is very similar to a previous specification from which we have already derived an executable program. A well established methodology which supports the reuse of program derivations from specifications is the one based on *schema transformations.*

A program schema is a syntactic abstraction of many concrete programs, which are called *instances* of that schema. When a program P_1 is transformed into a program P_2, possibly using the rule-based approach, in order to reuse this derivation, we may promote the transformation from P_1 to P_2, to a transformation from a schema S_1 to a schema S_2, such that P_1 and P_2 are instances of S_1 and S_2, respectively, according to the same substitution.

The schema-based approach to program transformation has been used in logic programming and also in imperative and functional programming (see, for instance, [2, 26, 39, 40, 54]).

Some methodologies for developing logic programs using program schemata have been proposed by several authors [18, 19, 28]) and various examples of schema transformations can be found in [8, 9, 16, 53, 62]. The schema transformations presented in those papers allow, for instance, for i) the transformation of left-recursive programs into right-recursive ones, ii) the reduction of nondeterminism in generate-and-test programs, and iii) the efficient and-parallel execution of logic programs.

Informally, we may say that a program schema is a program where some portions are left unspecified. The formal definition of a program schema requires, however, the introduction of a language where one can denote higher-order objects, such as variables ranging over predicates and goals. The reader may refer to [18, 25, 62] for some proposals of such languages in the case of logic programs. In this section, we simply use for program schemata the same syntax usually adopted for programs, except that we stipulate that variables may also occur in predicate positions, and in those positions they may range over predicate names.

Let us now consider a simple example where we show how schema transformations can be discovered and how they can be used during program derivation. Let us consider the following program *L-Fact* for evaluating the factorial function (for simplicity we omitted the clauses for *times*):

1. $\mathit{fact}(0, s(0)) \leftarrow$
2. $\mathit{fact}(s(X), Y) \leftarrow \mathit{fact}(X, Z),\ \mathit{times}(s(X), Z, Y)$

Program *L-Fact*: Left-recursive factorial.

Let us suppose that from this program we have derived the following program *R-Fact*:

3. $\mathit{fact}(X, Y) \leftarrow g(0, s(0), X, Y)$
4. $g(X, Y, X, Y) \leftarrow$
5. $g(X, Y, X1, Y1) \leftarrow \mathit{times}(s(X), Y, Z),\ g(s(X), Z, X1, Y1)$

Program *R-Fact*: Right-recursive factorial.

To justify the above transformation we do not need any property of the predicate *times*. Indeed, if we use the unfold/fold rules for performing this derivation, no unfolding step w.r.t. *times* is required. Thus we may apply the above transformation to every program which is an instance of the following schema L_1:

6. $F(0, s(0)) \leftarrow$
7. $F(s(X), Y) \leftarrow F(X, Z),\ T(s(X), Z, Y)$

Schema L_1: Left-recursive computation on natural numbers.

and produce the corresponding instance of the following schema R_1:

8. $F(X, Y) \leftarrow G(0, s(0), X, Y)$
9. $G(X, Y, X, Y) \leftarrow$
10. $G(X, Y, X1, Y1) \leftarrow T(s(X), Y, Z),\ G(s(X), Z, X1, Y1)$

Schema R_1: Right-recursive computation on natural numbers.

where F, G, and T range over distinct predicate symbols and the clauses for the predicate denoted by T are left unspecified. This schema transformation which is represented as $L_1 \rightarrow R_1$, can be applied during any subsequent program development by instantiating the predicate variables to predicate names and providing some clauses for the concrete predicate corresponding to T.

The usefulness of a schema transformation depends on its generality. However, to find schema transformations which are very general, is not an easy task. The reader may look at [18, 39, 54] for some methodologies which help designing general schema transformations. For example, our transformation of program *L-Fact* to program *R-Fact* may also be viewed as an instance of the following transformation from the program schema L_2 to the program schema R_2, where variables in argument positions range over tuples. This schema transformation (proposed in [9]) is more general than $L_1 \rightarrow R_1$ because it abstracts away from the data domain where the computations occur.

11. $P(X) \leftarrow A(X)$
12. $P(X) \leftarrow P(Y),\ B(Y, X)$

Schema L_2: Left-recursive computation.

13. $P(X) \leftarrow A(X1),\ G(X1, X)$
14. $G(X, X) \leftarrow$
15. $G(X, Y) \leftarrow B(X, Z),\ G(Z, Y)$

Schema R_2: Right-recursive computation.

The schema transformation $L_2 \rightarrow R_2$ can be applied to a much larger class of programs than the schema transformation $L_1 \rightarrow R_1$. In particular, we may use it for transforming the following program which is an instance of L_2 and computes a path PA in a directed graph from a given initial node A to any final node:

16. $path(A, PA) \leftarrow a(A, PA)$
17. $path(A, PA) \leftarrow path(B, PB),\ b(B, PB, A, PA)$
18. $a(A, PA) \leftarrow final(A),\ PA = [\,]$
19. $b(B, PB, A, PA) \leftarrow PA = [(A, B)|PB],\ edge(A, B)$

where $path(A, P)$ holds iff *either* $P = [\,]$ and $final(A)$ holds *or* for some $B_0, B_1, \ldots, B_n$, with $n \geq 0$, $P = [(A, B_0),\ (B_0, B_1),\ \ldots, (B_{n-1}, B_n)]$ and $final(B_n)$ holds. The corresponding instance of the schema R_2 is the program:

20. $path(A, PA) \leftarrow a(B, PB),\ g(B, PB, A, PA)$
21. $g(A, PA, A, PA) \leftarrow$
22. $g(A, PA, B, PB) \leftarrow b(A, PA, C, PC),\ g(C, PC, B, PB)$

together with clauses 18 and 19.

Notice, however, that the process of matching concrete programs to program schemata may be quite complex if the schemata we consider are very general. For instance, in the case of the program *L-Fact*, in order to show that it is an instance of schema L_2 we have first to transform it into the following program *L-Fact*1:

23. $fact(M, F) \leftarrow a(M, F)$
24. $fact(M, F) \leftarrow fact(X, G),\ b(X, Z, M, Y)$
25. $a(0, s(0)) \leftarrow$
26. $b(X, Z, M, Y) \leftarrow M = s(X),\ times(M, Z, Y)$

Program *L-Fact*1: Left-recursive factorial.

The problem of matching a given program to a given schema also depends on the formalism used and, in particular, it has been formalized as a matching problem of second-order terms [26].

Sometimes schema transformations may have associated applicability conditions which express constraints on the variables occurring in the schemata. In these cases a schema transformation is of the form: if p then $S_1 \rightarrow S_2$, and we may need theorem proving techniques to check whether or not the condition p holds.

Let us now briefly consider the problem of validating schema transformations obtained by abstraction from concrete program equivalences. The correctness proof of a schema transformation, say $S_1 \rightarrow S_2$, w.r.t. a given semantics amounts to show that if P_1 and P_2 are programs obtained by instantiating S_1 and S_2, respectively, via the same substitution then P_1 and P_2 are equivalent w.r.t. that semantics. Among the various methods for showing the schema correctness we recall here those based on denotational semantics [26, 39] and those based on the unfold/fold rules [31, 47].

Having constructed a large catalogue of schema transformations, we may perform complex transformations in one step only, and this may increase the efficiency of the program derivation process. However, for an effective use of that catalogue of transformations one has to address a few problems, like, for instance, the problem of the space requirements because the catalogue of schemata may be very large, and the problem of possible conflicts, in the sense that many schema transformations may be applicable at the same time in overlapping portions of the program at hand. Thus, strategies are needed to choose at each step the most convenient schema transformation to be applied. Some strategies, in the case of functional and logic programs may be found, for instance, in [18, 39, 54, 62].

4 Partial Evaluation

Partial evaluation is a transformation technique that specializes programs by exploiting some information about the context in which they run. In particular, partial evaluation (see [27] for a comprehensive account) may be used for deriving efficient logic programs by taking advantage of some partial knowledge about the input data.

In the past, partial evaluation has mainly been used for compiler generation, starting from the observation that a compiled program can be viewed as the result of partially evaluating an interpreter w.r.t. a source program [21]. Currently, partial evaluation has a growing impact on software engineering as a tool for program specialization which supports software reuse by program adaptation [13].

Partial evaluation of logic programs (which is also called partial deduction) has been introduced in [29] and then formalized in [36]. It has been fruitfully applied to specialize meta-interpreters [22, 49, 55, 57, 58], that is, logic programs which behave as interpreters of logic programs. Specialization of meta-interpreters forms the basis of an important technique which can be used to enhance logic programming. Indeed, sophisticated functionalities and evaluation mechanisms can be added to logic languages by providing suitable meta-interpreters, and then efficient programs can be derived by specializing at compile-time a given meta-interpreter to some given input programs.

The basic technique for partial evaluation was presented in [23]. We now illustrate how this technique can be viewed as a sequence of applications of the transformation rules we have listed in Section 2.

Let us assume that we want to partially evaluate a given program P_0 w.r.t. a given goal G which is the conjunction of the atoms g_k, for $k = 1, \ldots, K$. Then we may proceed as follows. We first introduce a set A_0 of K definition clauses of the form: $H_k \leftarrow g_k$, for $k = 1, \ldots, K$. Then, from this program we construct a sequence of programs of the form: $P_0 \cup A_i \cup F_i$, for $i \geq 0$, with

$F_0 = \{\}$. In order to construct the program $P_0 \cup A_{i+1} \cup F_{i+1}$ from the program $P_0 \cup A_i \cup F_i$, for $i \geq 0$, we unfold once or more times each clause in A_i and we derive a new program $P_0 \cup U_i \cup F_i$. We then introduce a new set A_{i+1} of definitions whose bodies are *generalizations* of the atoms occurring in the bodies of the clauses in U_i and we fold all these atoms using the definitions in $A_0 \cup \ldots \cup A_{i+1}$, thereby obtaining the program $P_0 \cup A_{i+1} \cup F_{i+1}$. Partial evaluation terminates for $i = n$ iff all atoms occurring in the bodies of the clauses of U_n can be folded using the definitions occurring in $A_0 \cup \ldots \cup A_n$, that is, A_{n+1} is empty because no new definition is needed for folding the atoms in the bodies of U_n. The final program is $P_0 \cup F_n$, and P_0 can be dropped if we are interested only in queries relative to predicates defined by A_0, which are the partially evaluated queries, because during the evaluation of these queries, only calls of predicates in F_n may be generated.

The total correctness w.r.t. the least Herbrand model semantics of the transformations realized by the above partial evaluation technique, is an easy consequence of the correctness of the transformation rules presented in Section 2. Moreover, it can be shown that, since all definitions have precisely one atom in their body, partial evaluation also preserves the finite failure set [52]. Correctness results of partial evaluation are also available in the literature for logic programming languages with negation, constraints, and other extensions of definite programs. Various other semantics have also been considered, and we refer to [36, 41] for further details.

In the partial evaluation technique as we have described above, we have not fully specified how to perform the sequences of applications of each transformation rule. In particular, for controlling the applications of the unfolding rule, we need to select, at each step an atom in the body of the clause to be unfolded, and we also need to decide when to terminate the sequence of unfolding steps, which may otherwise be infinite. Another important control issue is related to the choice of suitable generalizations of the atoms in U_i which are bodies of the definitions to be used for the folding steps. This issue is particularly important because on the one hand we may want to introduce definition clauses with very general bodies so that we can terminate the partial evaluation process, and on the other hand the bodies of these definitions should not be too general because otherwise we may prevent effective program specialization, because too many data structures have been generalized to variables.

Various control strategies for unfolding and generalization have been studied in [3, 37, 45] and also implemented in various partial evaluation systems for logic programs, such as SP [23], ProMiX [33], Logimix [38], PADDY [44], and Mixtus [50].

Let us now see in action the partial evaluation technique in an example which consists in specializing a general parser for regular languages to a given regular expression. This example shows that partial evaluation may also be used as a tool for program compilation, because the specialized parser corresponds to the finite automaton accepting the language denoted by the given regular expression.

We are given the following general parser for regular languages over the alphabet $\{a, b\}$, where $accepts(A, S, D)$ holds iff i) the string S is the concatenation of a string C and the string D, and ii) C belongs to the language accepted by the regular expression A:

1. $accepts(A, [A|B], B) \leftarrow symbol(A)$
2. $accepts((A \cdot B), C, D) \leftarrow accepts(A, C, E),\ accepts(B, E, D)$
3. $accepts(A + B, C, D) \leftarrow accepts(A, C, D)$
4. $accepts(A + B, C, D) \leftarrow accepts(B, C, D)$
5. $accepts(A^*, B, B) \leftarrow$

6. $accepts(A^*, B, C) \leftarrow accepts((A \cdot A^*), B, C)$
7. $symbol(a) \leftarrow$
8. $symbol(b) \leftarrow$

and we want to specialize it to the regular expression $(a \cdot b) + a^*$. Thus we introduce the following clause:

9. $new1(A) \leftarrow accepts((a \cdot b) + a^*, A, [\,])$

and we have: $A_0 = \{\text{clause } 9\}$.

During partial evaluation we will follow the strategy of performing the unfolding of the leftmost atom of the clause at hand until either unfolding is no longer possible or we get an instantiation of the head which exposes a symbol in the given alphabet $\{a, b\}$. Thus, by unfolding clause 9 we get the set of clauses U_0, made out of the following clauses:

10. $new1([a|B]) \leftarrow accepts(b, B, [\,])$
11. $new1([\,]) \leftarrow$
12. $new1([a|C]) \leftarrow accepts(a^*, C, [\,])$

Now clause 11 cannot be unfolded and we do not unfold clauses 10 and 12 either, because in their heads both arguments $[a|B]$ and $[a|C]$ expose an a. We then introduce two new predicates $new2$ and $new3$ whose definitions are derived from the bodies of the clauses in U_0 and we get, after the corresponding folding steps, the following clauses:

13. $new2(B) \leftarrow accepts(b, B, [\,])$
14. $new3(C) \leftarrow accepts(a^*, C, [\,])$
15. $new1([a|B]) \leftarrow new2(B)$ (by folding clause 10 using clause 13)
16. $new1([a|C]) \leftarrow new3(C)$ (by folding clause 12 using clause 14)

We have that $A_1 = \{\text{clause } 13, \text{ clause } 14\}$ and $F_1 = \{\text{clause } 15, \text{ clause } 16\}$. By unfolding the clauses in A_1 we get the set U_1 made out of the following clauses:

17. $new2([b]) \leftarrow$
18. $new3([\,]) \leftarrow$
19. $new3([a|D]) \leftarrow accepts(a \cdot a^*, D, [\,])$

Now clause 19 can be folded using clause 14 in A_1 and we get:

20. $new3([a|D]) \leftarrow new3(D)$

Thus, no new definition is needed for folding the clauses in U_1 and we stop the partial evaluation process. The final program is made out of clauses 11, 15, 16, 17, 18, and 20. The reader may see that the three new predicates indeed correspond to the three states of a finite automaton which recognizes $(a \cdot b) + a^*$.

We should now remark that the derived finite automaton does not have a minimal number of states and it is non-deterministic. These facts show that the basic techniques for partial evaluation have some limitations and they are not always able to fully exploit the available knowledge about the input data. The limitations of partial evaluation have also been studied from a theoretical point of view and it has been shown that, under some suitable hypothesis, this technique can only achieve linear speedups [27, Chapter 6].

A considerable amount of current research work is devoted to the enhancement of partial evaluation towards more powerful techniques for program specialization. To this aim, recently it has been suggested i) to deal with the specialization of conjunctions and disjunctions of goals (instead of atomic goals only) by incorporating more powerful transformation rules [34, 43], and ii) to specialize programs w.r.t. a class of input data satisfying some properties (instead of particular input data) by using either more powerful transformation rules [4, 42] (possibly based on theorem proving techniques) or abstract interpretation [24].

5 Program Transformation and Logic-based Software Engineering: Achievements and Future Developments

In the previous sections we have presented through some simple examples, some techniques for logic program transformation and we have shown that they may provide valuable tools for logic program development. In this section we would like to consider a wider perspective and illustrate that program transformation can fruitfully be used also for logic-based software engineering.

By considering the achievements of logic program transformation techniques over the last years, we want to briefly indicate how these techniques can be used for i) improving the efficiency of programs in logic-based systems, ii) supporting modularity and reuse during program development, and iii) providing general tools for knowledge representation and management.

Point i). In the field of program transformation several methods have been developed during the past years for improving the efficiency of logic programs. Various techniques have been proposed for the automatic introduction of sophisticated data structures into logic programs. Among others, we may mention, for instance, the difference lists [12], whereby expensive operations can be performed in an efficient way, and in particular, list concatenation can be done in constant time. Other techniques have been developed with the aim of eliminating data structures which are not relevant for the final answers and are used only for storing intermediate information [48]. In the literature we can also find methods for improving the efficiency of logic programs by transforming them into equivalent programs which incorporate sophisticated strategies to reduce the amount of nondeterminism during the search for a successful derivation [10]. Similarly, transformation techniques may increase program efficiency by either removing recursion in favour of iteration [14], or adding annotations, such as cuts, to programs so that the evaluators may improve memory usage [17, 51], or deriving particular forms of programs, like binary programs [60], which allow for compilers with high levels of performance.

We think that a major challenge for the near future is to integrate these methods so that they could form a basis for a new generation of optimizing compilers for logic programs. These future compilers could also take advantage of the computer architecture by improving, for instance, the parallel executability of concurrent subtasks (see [16] for work in this direction).

We also think that program transformation may be usefully applied to construct optimizing compilers for enhancements of logic languages which incorporate various important extensions, such as concurrency, constraints, higher-order predicates, and objects.

Point ii). Logic program transformation techniques have been mainly designed for small programs within single software modules. When large software systems have to be developed, program transformation should support methodologies for modular software development. To this aim, we need to further develop the transformational methods which are currently available so to allow the user to compose many program modules together. Such methods may also be integrated with other methods for logic program composition such as those described in [7, 32].

Another important issue to be addressed when developing large software systems, is the reusability and adaptation of part of the software modules for different purposes and in different contexts. To allow for maximal reusability, our software should be as abstract and parametric as possible. In particular, program modules which abstract away from the concrete implementation of the data structures or from the data which are specific to the context at hand, are most suitable for reuse and adaptation. Having these abstract modules, we can then apply the techniques developed in the field of program specialization which we have briefly described in Section 4, for automatically instantiating and customizing software modules (see [30] for some initial work in this direction).

Point iii). In Section 4 we have already mentioned that program transformation can be applied to specialize meta-interpreters. We believe that the potentiality of the interaction among knowledge representation, knowledge management, meta-programming, and program transformation still has to exploited. In particular, logic programming supports language enhancement through meta-interpretation (see also [56]): new language features may be added to a kernel language by writing an interpreter of the enhanced language using the kernel language itself. Program transformation can then be used to improve efficiency by specializing the interpreter for each given object program.

We believe this approach should be further pursued for the development of enhanced languages through which we are able to describe complex forms of reasoning and knowledge manipulation, such as abductive reasoning, epistemic reasoning, reflective reasoning, temporal reasoning, etc. Some work on this direction has already been done in [61], where a transformation of abductive logic programs into normal logic programs is presented. This transformation, however, has not been done through specialization of meta-interpreters.

Finally, we want to say a few words on the inverse interaction between logic-based software engineering and program transformation, and in particular about the role that the various techniques for knowledge engineering can play for improving the available systems for the automatic development and transformation of logic programs. Indeed, many such systems have to store and manipulate information about programs, like, for instance, their derivation histories, the predicate definitions which have been introduced during their derivations, the syntactic properties they enjoy, etc. Thus, advanced tools for storing, manipulating, and restructuring information of that kind can be useful for the development of high performance transformation systems.

Also the various techniques for meta-programming and meta-language definition which have been proposed for knowledge representation (see, for instance, [1]) can fruitfully be applied when developing automatic program derivation systems, because program transformers are particular meta-programs which take programs as input and produce new programs as output. The process of deriving transformation histories from previous program developments and then replaying those histories starting from analogous initial specifications, can also be viewed as the execution of a suitable meta-program on an input data program. In order to improve program efficiency it may be worthwhile to apply several transformation histories in succession and thus, to have meta-programs which act on meta-programs. This situation occurs when we consider 'incremental' transformation techniques, that is, techniques whose iterative application may monotonically improve the quality of the derived programs. More research and experimentation needs to be done in this direction, and in particular, for the development of integrated transformation systems in which strategies at higher levels direct the application of strategies at lower levels, and the outcome of previous program derivations is used for the automatic improvement of the available strategies. We think that such issues constitute some of the major challenges for the future of program transformation systems.

Acknowledgements

We would like to thank N. Fuchs for his encouragement and helpful discussions.

References

[1] K.R. Apt and F. Turini, editors. *Meta-Logics for Logic Programming.* The MIT Press, 1995.

[2] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[3] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, USA*, pages 343–358. The MIT Press, 1990.

[4] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[5] A. Bossi, N. Cocco, and S. Etalle. Transforming left-terminating programs: The reordering problem. In M. Proietti, editor, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '95, Utrecht, The Netherlands.*, pages 33–45. Springer-Verlag, 1996.

[6] R. S. Boyer and J. S. Moore. Proving theorems about Lisp functions. *Journal of the ACM*, 22(1):129–144, 1975.

[7] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J. W. Lloyd, editor, *Proceedings of the Symposium on Computational Logic*, pages 117–134. Springer-Verlag, 1990.

[8] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *Journal of Logic Programming*, 4:345–359, 1987.

[9] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(1):115–134, 1991.

[10] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.

[11] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[12] K. L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proceedings Information Processing '77*, pages 939–944. North-Holland, 1977.

[13] C. Consel. Program adaptation based on program transformation. Position paper for the MIT Workshop on Strategic Directions of Computing Research, June 1996.

[14] S. K. Debray. Optimizing almost-tail-recursive Prolog programs. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, Lecture Notes in Computer Science 201, pages 204–219. Springer-Verlag, 1985.

[15] S. K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA*, SIGPLAN Notices, 23, (7), pages 297–307, 1988.

[16] S. K. Debray and M. Jain. A simple program transformation for parallelism. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 305–319. The MIT Press, 1994.

[17] S. K. Debray and D. S. Warren. Functional computations in logic programs. *ACM TOPLAS*, 11(3):451–481, 1989.

[18] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation*, 15:775–805, 1993.

[19] N. E. Fuchs and M. P. J. Fromherz. Schema-based transformations of logic programs. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '91, Manchester, U.K.*, pages 111–125. Springer-Verlag, 1992.

[20] N. E. Fuchs and R. Schwitter. Specifying logic programs in controlled natural language. In *Workshop on Computational Logic for Natural Language Processing, Proceedings CLNLP '95*, 1995.

[21] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[22] J. P. Gallagher. Transforming programs by specializing interpreters. In *Proceedings Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, 1986.

[23] J. P. Gallagher. Tutorial on specialization of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[24] J. P. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In *Proceedings of LoPSTr'92, Manchester, U.K.*, pages 151–167. Springer–Verlag, 1993.

[25] T. S. Gegg-Harrison. Representing logic program schemata in λ-Prolog. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 467–481. P, 1995.

[26] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[27] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[28] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and techniques for Prolog programming. TR 89-170, Case Western Reserve University, 1989.

[29] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA*, pages 255–267, 1982.

[30] H. J. Komorowski. Towards synthesis of programs in the partial deduction framework. In *Proceedings of the Workshop on Automating Software Design, Detroit, MI*, pages 138–143. Kestrel Institute, 1989.

[31] L. Kott. Unfold/fold program transformation. In *Algebraic Methods in Semantics*, pages 411–434. Cambridge University Press, 1985.

[32] A. Lakhotia and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2 & 3):211–226, 1988.

[33] A. Lakhotia and L. Sterling. ProMix: A Prolog partial evaluation system. In L. Sterling, editor, *The Practice of Prolog*, pages 137–179. The MIT Press, 1990.

[34] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. Report CW 225, K.U. Leuven, Belgium, 1996.

[35] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.

[36] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[37] B. Martens, D. De Schreye, and M. Bruynooghe. Sound and complete partial deduction with unfolding based on well-founded measures. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 473–480. Ohmsha Ltd., IOS Press, 1992.

[38] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '92, Manchester, U.K.*, Workshops in Computing, pages 214–227. Springer-Verlag, 1993.

[39] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[40] M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Mass., USA*, pages 119–127, 1970.

[41] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[42] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation.* Springer-Verlag, 1996. (To appear).

[43] A. Pettorossi, M. Proietti, and S. Renault. Enhancing partial deduction via unfold/fold rules. In *Logic Program Synthesis and Transformation, Proceedings of LOPSTR '96, Stockholm, Sweden.* Springer-Verlag, 1996. (To appear).

[44] S. Prestwich. Online partial deduction of large programs. In *Proceedings ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 111–118. ACM Press, 1993.

[45] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16(1–2):123–161, 1993.

[46] M. Proietti and A. Pettorossi. Completeness of some transformation strategies for avoiding unnecessary logical variables. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming (ICLP '94), S. Margherita Ligure, Italy, June 13–18*, pages 714–729. The MIT Press, 1994.

[47] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Logic Program Synthesis and Transformation, Proceedings of LOPSTR '93, Louvain-la-Neuve, Belgium*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994.

[48] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[49] S. Safra and E. Shapiro. Meta interpreters for real. In H. J. Kugler, editor, *Proceedings Information Processing 86*, pages 271–278. North-Holland, 1986.

[50] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.

[51] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their application to Prolog optimization. In *Proceedings of the International Symposium on Logic Programming, Boston*, pages 200–207. IEEE Computer Society Press, 1985.

[52] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[53] H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *Proceedings of the International Symposium on Logic Programming, San Francisco, USA*, pages 215–223. IEEE Press, 1987.

[54] D. R. Smith. Automating the design of algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development, IFIP TC2/W.G 2.1 State-of-the-Art Report*, Lecture Notes in Computer Science 755, pages 324–354. Springer-Verlag, 1993.

[55] L. Sterling. Incremental flavour-mixing of meta-interpreters for expert system construction. In *Proceedings 3rd International Symposium on Logic Programming, Salt Lake City, Utah, USA*, pages 20–27. IEEE Press, 1986.

[56] L. S. Sterling and L. Ü. Yalcinalp. Logic programming and software engineering. *Knowledge Engineering Review*, 1996.

[57] A. Takeuchi. Affinity between meta-interpreters and partial evaluation. In H. J. Kugler, editor, *Proceedings of Information Processing '86*, pages 279–282. North-Holland, 1986.

[58] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta-programming. In H. J. Kugler, editor, *Proceedings of Information Processing '86*, pages 279–282. North-Holland, 1986.

[59] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[60] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Małuszynski, editors, *Proceedings PLILP '90*, pages 159–173. Springer-Verlag, 1990.

[61] F. Toni and R. Kowalski. Reduction of abductive logic programs to normal logic programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 367–381. The MIT Press, 1995.

[62] W. W. Vasconcelos and N. E. Fuchs. An opportunistic approach for logic program analysis and optimization using enhanced schema-based transformations. In M. Proietti, editor, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '95, Utrecht, The Netherlands.*, pages 174–188. Springer-Verlag, 1996.