Reducing Nondeterminism while Specializing Logic Programs

Alberto Pettorossi
Department of Informatics,
Systems, and Management
University of Roma Tor Vergata
00133 Roma, Italy
adp@iasi.rm.cnr.it

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

Sophie Renault
Department of Informatics,
Systems, and Management
University of Roma Tor Vergata
00133 Roma, Italy
renault@iasi.rm.cnr.it

Abstract

Program specialization is a collection of program transformation techniques for improving program efficiency by exploiting some information available at compile-time about the input data. We show that current techniques for program specialization based on partial evaluation do not perform well on nondeterministic logic programs. We then consider a set of transformation rules which extend the ones used for partial evaluation, and we propose a strategy to direct the application of these extended rules so to derive very efficient specialized programs. The efficiency improvements which may even be exponential, are achieved because the derived programs are semi-deterministic and the operations which are performed by the initial programs in different branches of the computation trees, are performed in the specialized programs within single branches. We also make use of mode information to guide the unfolding process and to reduce nondeterminism. To exemplify our technique, we show that we can automatically derive very efficient semi-deterministic matching programs and semi-deterministic parsers for regular languages. The derivations we have performed could not have been done by previously known partial evaluation techniques.

1 Introduction and Preliminary Example

The idea of *program specialization* [8] has been applied in several areas of computer science, and it has found its application also in the case of logic programming [5]. It is well known that by program specialization one may derive from old programs new, more efficient programs by taking into account available information about the input data. In this paper we follow a *rule-based* approach to program specialization. In particular, we consider definite logic programs [11] and we propose program specialization techniques using unfold/fold transformation rules [2, 18]. In our approach, the process of program specialization can be viewed as the construction of a sequence of programs, say $P_0, \ldots, P_s$, where P_0 is the program to be specialized and every program of this sequence is obtained from the previous one by applying a transformation rule. P_s is the final specialized program.

Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997, ACM Press, 1997, pp. 414–427.

Now *partial evaluation* of logic programs [12] can be viewed as a particular technique of logic program specialization when the following three transformation rules are used (see, for instance, [16] for a rule-based approach to partial evaluation of logic programs).
1. *Atomic Definition Rule.* We introduce a clause of the form $newp(X_1, \ldots, X_h) \leftarrow A$ where *newp* is a new predicate, A is an atom whose predicate occurs in the initial program P_0, and $X_1, \ldots, X_h$ are the variables occurring in A.
2. *Unfolding Rule.* We replace a clause, say C, by those obtained from it by performing an SLD-resolution step w.r.t. a selected atom in the body of C. If no SLD-resolution step is possible w.r.t. the selected atom then clause C is deleted from the program.
3. *Atomic Folding Rule.* Let $N \leftarrow A$ be a clause introduced by the definition rule, and C a clause of the form: $H \leftarrow F, A\theta, G$ for some sequences of atoms F and G, and some substitution θ. Let us also assume that C is a clause which has not been introduced by the definition rule. By folding we replace C by the clause $H \leftarrow F, N\theta, G$.

Although partial evaluation has been widely recognized as a very useful technique, it has some intrinsic limitations due to the use of the above transformation rules. We would like to point out some of these limitations and in the rest of the paper we present a transformation technique based on more powerful rules which allow us to overcome these limitations.

The usual partial evaluation techniques apply the transformation rules according to a strategy which can be expressed as follows. In this strategy we leave unspecified the subsidiary strategies for unfolding and generalization.

Partial Evaluation Strategy. Given a program P_0 and an atomic goal A w.r.t. which we want to specialize P_0, we introduce by the atomic definition rule the following new clause, called *specializing clause*, D_0: $newp(X_1, \ldots, X_h) \leftarrow A$, where $newp$ is a new predicate and $X_1, \ldots, X_h$ are the distinct variables occurring in A. Clause D_0 is then unfolded, using the clauses in P_0, according to a given *unfolding strategy* which indicates the atom w.r.t. which the unfolding rule should be applied and when to stop the unfolding process.

We then consider the set B of all atoms occurring in the bodies of the clauses derived by unfolding. If all atoms in B can be folded (that is, they are all instances of A) then we stop partial evaluation, otherwise we introduce, by applying again the atomic definition rule, some new clauses $D_1, \ldots, D_n$ such that for each atom E in B there exists a clause in $D_0, D_1, \ldots, D_n$ whose body is a variant or a generalization of E. This application of the atomic definition rule is performed according to a so-called *generalization strategy.* We thus fold all atoms in B using clauses in $D_0, D_1, \ldots, D_n$.

Partial evaluation proceeds by transforming each clause $D_1, \ldots, D_n$ following the same steps we have described for D_0. The whole process stops when each atom in the body of every clause derived by unfolding can be folded by using one of the clauses introduced by the atomic definition rule. □

One major weakness of the partial evaluation strategy is that it divides the bodies of the clauses derived by unfolding into single atoms, thereby hindering further program specializations, as we now indicate through an example.

Let us consider the following initial program for checking whether or not the pattern P occurs in the string S:

1.1 $match(P, S) \leftarrow app(L, P, X), app(X, R, S)$
1.2 $app([\,], Y, Y) \leftarrow$
1.3 $app([A|X], Y, [A|Z]) \leftarrow app(X, Y, Z)$

Suppose that we want to specialize this program w.r.t. the goal $match([a, a, b], S)$. Then, the partial evaluation strategy works by introducing, by atomic definition, the specializing clause:

2. $sp\_match(S) \leftarrow match([a, a, b], S)$

Now the unfolding rule is applied to clause 2 according to a given unfolding strategy, whose detailed description is not relevant here. Let us assume that we thus derive the following two clauses:

3. $sp\_match([a|S]) \leftarrow app([a, b], R, S)$
4. $sp\_match([C|S]) \leftarrow app(L, [a, a, b], X), app(X, R, S)$

Now the application of the partial evaluation strategy proceeds by using a given generalization strategy which introduces some new clauses such that each *app* atom in the body of clauses 3 and 4 can be folded by distinct folding steps. Then those new clauses are transformed one at a time. Unfortunately, these independent transformations do not allow us to exploit the interactions among the *app* atoms. Indeed, suppose that by the generalization strategy we introduce one clause for each *app* atom as follows:

5. $sp\_app1(R, S) \leftarrow app([a, b], R, S)$
6. $sp\_app2(L, X) \leftarrow app(L, [a, a, b], X)$
7. $sp\_app3(X, R, S) \leftarrow app(X, R, S)$

so that by folding the *app* atoms in clauses 3 and 4 we get

8. $sp\_match([a|S]) \leftarrow sp\_app1(R, S)$
9. $sp\_match([C|S]) \leftarrow sp\_app2(L, X), sp\_app3(X, R, S)$

By using clauses 8 and 9 we still have nondeterministic computations, because the atom $sp\_match([a|S])$ unifies with both clause heads. Moreover, when clause 9 is used, the intermediate list X is generated and also the bindings for L and R are unnecessarily computed.

In contrast, by using the transformation technique we introduce in this paper, one can derive specialized programs and avoid both unnecessary nondeterministic computations and the generation of redundant data structures. This improvement is achieved by making use of powerful definition and folding rules (similar to those in [6]), and also by taking advantage of a rule, called *case split*, which we introduce in Section 3.

In our example, the definition, folding, and case split rules work as follows. We consider clauses 3 and 4 and we notice that a ground call of the form $sp\_match([C|S])$ is unifiable with the heads of both clauses if and only if $C = a$. This fact suggests that, in order to avoid nondeterminism, one may distinguish the two cases where $C = a$ and $C \neq a$. Thus, as an application of the case split rule we replace clause 4 by the following clauses:

4.1 $sp\_match([a|S]) \leftarrow app(L, [a, a, b], X), app(X, R, S)$
4.2 $sp\_match([C|S]) \leftarrow C \neq a, app(L, [a, a, b], X), app(X, R, S)$

Now, by using an extended definition rule (see Section 3) so that a single predicate may be defined by more than one clause whose body may contain more than one atom, we introduce the following two new predicates:

10. $new1(S) \leftarrow app([a, b], R, S)$
11. $new1(S) \leftarrow app(L, [a, a, b], X), app(X, R, S)$
12. $new2(S) \leftarrow app(L, [a, a, b], X), app(X, R, S)$

We may now apply the extended folding rule we consider in this paper and by folding clauses 3 and 4.1 using clauses 10 and 11, and by folding clause 4.2 using clause 12, we get, respectively:

13. $sp\_match([a|S]) \leftarrow new1(S)$
14. $sp\_match([C|S]) \leftarrow C \neq a,\ new2(S)$

Now using clauses 13 and 14, instead of clauses 3 and 4, the computation is *deterministic*, in the sense that for ground calls of the form $sp\_match([C|S])$ only one clause may be used. Moreover, no intermediate or unnecessary bindings are constructed.

However, we still have nondeterministic computations and we still construct intermediate or unnecessary bindings when the clauses 10, 11, and 12 are used. Thus, we now need to transform these clauses for avoiding these drawbacks. To this aim we may perform unfolding, case split, extended definition, and extended folding steps, in this order, following a strategy similar to the one described above in the case of the derivation of clauses 13 and 14 from clauses 3 and 4. This strategy which enhances the partial evaluation strategy, is described in Section 5. In that section we also present an example of specialization of a multiple string matching program which generalizes the example we have now given.

We would like to point out some important features of our proposed specialization technique which we have seen in action in the derivation of clauses 13 and 14.
1) During program specialization we are able to introduce new predicates defined by *several* clauses whose bodies are possibly *non-atomic* goals (see clauses 10 and 11 defining predicate $new1$). By folding using definition clauses of this form we can reduce nondeterminism in the specialized program. This extended folding rule also allows us to avoid intermediate data structures.
2) In order to reduce the nondeterminism in programs, we make use of a case split rule. Case split derives two or more mutually exclusive instances of a given clause by introducing negative tests (see clauses 4.1 and 4.2).
3) In order to evaluate negative tests of the form $C \neq a$ (see clause 4.2) by using negation as failure, the variable C should be bound at runtime to a non-variable term. In general, the suitable instantiations which are necessary for evaluating inequalities, can be ensured by using the concept of predicate *modes* [3]. In this paper we will make use of this concept which will also allow us to derive specialized programs which are efficient because they are *semi-deterministic* (see Section 4). Indeed, in semi-deterministic programs at most one non-unit clause may successfully be used for resolving any predicate call which satisfies the given mode.

Our main contribution consists in the proposal of an automatic technique which makes use of: 1) the definition and folding rules w.r.t. several clauses, 2) the case split rule, and also 3) the modes of the predicates for guiding the application of the transformation rules so to derive specialized programs with reduced nondeterminism. These three features characterize our contribution w.r.t. other recent proposals for enhancing partial evaluation by allowing more sophisticated rules or performing global program analysis based on abstract interpretation [5, 9].

This paper is structured as follows. In Section 2 we show through an example that the definition and folding rules we consider allow us to get exponential time speedups. These speedups are not possible if one uses partial evaluation alone, that is, the rules of atomic definition, unfolding, and atomic folding. In Section 3 we formally present our powerful transformation rules for program specialization and we discuss their correctness. In Section 4 we consider logic programs with modes and we introduce the concept of semi-deterministic programs. Modes provide an abstract interpretation framework which is used in the strategy presented in Section 5 for directing the application of the transformation rules. Finally, in Section 6 we present a brief overview of the related work and we discuss a few issues concerning the techniques we have proposed.

2 An Example of Exponential Speedup

Let us consider the following parser for context-free grammars:

1. $parse(G, [A], [A|X]\backslash X) \leftarrow terminal(A)$
2. $parse(G, [A], W) \leftarrow nonterminal(A),$
   $member(A \rightarrow Bs,\ G), parse(G, Bs, W)$
3. $parse(G, [A1, A2|As],\ W\backslash X) \leftarrow$
   $parse(G, [A1],\ W\backslash I), parse(G, [A2|As], I\backslash X)$
4. $member(A, [A|X]) \leftarrow$
5. $member(A, [B|X]) \leftarrow\ member(A, X)$

together with the unit clauses stating that 0 and 1 are terminals and s, u, v, and w are nonterminals. The first argument of *parse* is a context-free grammar, the second argument is a sentential form, that is, a list of terminal and nonterminal symbols, and the third argument is a word represented as a difference list, that is, for instance, a word $a_1 a_2 \ldots a_n$ is represented as $W\backslash X$, where W is the concatenation of $[a_1, a_2, \ldots, a_n]$ and X for some list X. We assume that context-free grammars are represented as lists of productions of the form $x \rightarrow y$, where x is a nonterminal symbol and y is a sentential form. We have that $parse(G, [A], W\backslash X)$ holds iff from the symbol A we can derive, according to the grammar G, the word represented as the difference list $W\backslash X$.

We want to specialize our parsing program w.r.t. the following regular grammar $g = [s \rightarrow [0, u],\ s \rightarrow [0, v],$ $s \rightarrow [0, w],\ u \rightarrow [0],\ u \rightarrow [0, u],\ u \rightarrow [0, v],\ v \rightarrow [0],$ $v \rightarrow [0, v],\ v \rightarrow [0, u],\ w \rightarrow [1],\ w \rightarrow [0, w]]$.

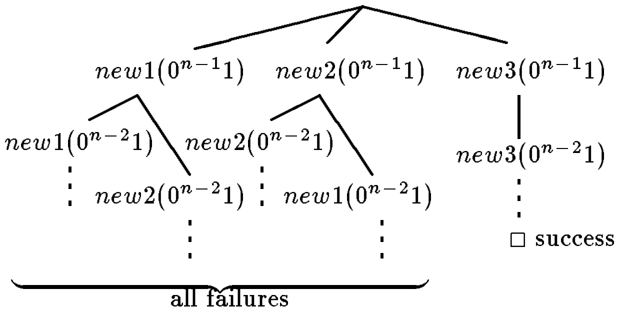

Figure 1: SLD-tree for $sp\_parse(0^n 1)$

To this aim we apply the partial evaluation technique, that is, we apply the atomic definition, unfolding, and atomic folding rules according to the strategy we have presented in Section 1. Thus, we first introduce the following specializing clause:

6. $sp\_parse(W) \leftarrow parse(g, [s], W\backslash[\,])$

After partial evaluation we get the following specialized program:

7. $sp\_parse([0|W]) \leftarrow new1(W)$
8. $sp\_parse([0|W]) \leftarrow new2(W)$
9. $sp\_parse([0|W]) \leftarrow new3(W)$
10. $new1([0]) \leftarrow$
11. $new1([0|W]) \leftarrow new1(W)$
12. $new1([0|W]) \leftarrow new2(W)$
13. $new2([0]) \leftarrow$
14. $new2([0|W]) \leftarrow new2(W)$
15. $new2([0|W]) \leftarrow new1(W)$
16. $new3([1]) \leftarrow$
17. $new3([0|W]) \leftarrow new3(W)$

which corresponds to a nondeterministic finite automaton. This program is not satisfactory because it takes $O(2^n)$ steps to accept a word of the form $0^n 1$ when evaluation is performed by using the depth-first search strategy. In order to see this, consider a goal of the form $sp\_parse(0^n 1)$, where $0^n 1$ denotes the list $[0, \ldots, 0, 1]$ with $n \geq 2$ occurrences of 0. The SLD-tree generated by this goal is depicted in Figure 1. In this tree the subtrees which are rooted in the goal $new1(0^{n-1}1)$ and $new2(0^{n-1}1)$ do not contain any success and they have exponential size w.r.t. n. Thus, if the SLD-tree is visited in a depth-first manner, then an exponential number of steps is required before terminating with success.

However, in the SLD-tree for $sp\_parse(0^n 1)$ there are many repeated subgoals. For instance, the goal $new1(0^{n-2}1)$ occurs both in the subtree rooted in the goal $new1(0^{n-1}1)$ and in the one rooted in the goal $new2(0^{n-1}1)$. Similarly, for $new2(0^{n-2}1)$.

Now we show that we can achieve exponential speedups by using more powerful definition and folding rules than the ones relative to partial evaluation. These exponential gains are achieved by deriving a program which corresponds to a deterministic finite automaton, so that redundant goals in the SLD-trees are avoided and a word of the form $0^n 1$ is accepted in $O(n)$ steps.

We first notice that a call of the form $sp\_parse([0|W])$ is evaluated nondeterministically because it is unifiable with the heads of the three clauses 7, 8, and 9. However, we can transform those clauses so that a call of the form $sp\_parse([0|W])$ is deterministic, that is, it is unifiable with the head of one clause only in the transformed program. To this aim we introduce the new predicate $new123$ by the following three clauses:

18. $new123(W) \leftarrow new1(W)$
19. $new123(W) \leftarrow new2(W)$
20. $new123(W) \leftarrow new3(W)$

Notice that this step uses an extended form of definition rule, because more than one clause has been introduced for the predicate $new123$. By folding clauses 7, 8, and 9 using clauses 18, 19, and 20, we get:

21. $sp\_parse([0|W]) \leftarrow new123(W)$

Notice that, also for this transformation step we need to extend the atomic folding rule.

Now any call of the form $sp\_parse([0|W])$ is deterministic, in the sense that it is unifiable with the head of clause 21 only. However, nondeterministic calls are not avoided, because the call $sp\_parse([0|W])$ generates the call $new123(W)$ which is unifiable with the heads of the three clauses 18, 19, and 20. Thus, we are now left with the problem of avoiding the nondeterminism due to clauses 18, 19, and 20. We proceed as follows. We unfold clauses 18, 19, and 20 and we get the following clauses:

22. $new123([0]) \leftarrow$
23. $new123([0|W]) \leftarrow new1(W)$
24. $new123([0|W]) \leftarrow new2(W)$
25. $new123([0]) \leftarrow$
26. $new123([0|W]) \leftarrow new2(W)$
27. $new123([0|W]) \leftarrow new1(W)$
28. $new123([1]) \leftarrow$
29. $new123([0|W]) \leftarrow new3(W)$

Clauses 25, 26, and 27 are equal to clauses 22, 24, and 23, respectively, and they may be deleted without modifying the least Herbrand model of the program. Moreover, we can fold clauses 23, 24, and 29 using clauses 18, 19, and 20 (also this step is an extended form of the folding rule because it uses more than one clause for folding) and we derive a recursive definition of $new123$. The final specialized program which corresponds to a deterministic finite automaton, is the following:

21. $sp\_parse([0|W]) \leftarrow new123(W)$
22. $new123([0]) \leftarrow$

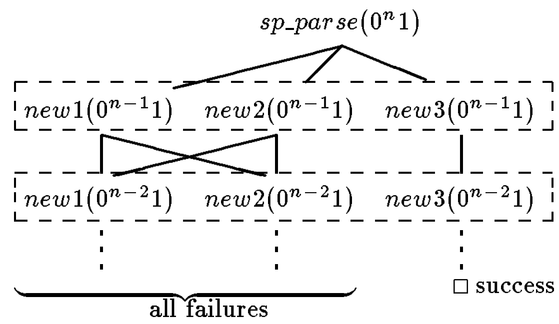

Figure 2: Graph of calls for $sp\_parse(0^n1)$

28. $new123([1]) \leftarrow$
30. $new123([0|W]) \leftarrow new123(W)$

This program does not include the clauses for *new*1, *new*2, and *new*3, and takes indeed $O(n)$ steps to accept the word 0^n1. Notice that our final program is not necessarily deterministic for calls of the form $sp\_parse(w)$, where w is a ground list, because a call of the form $new123([0])$ may be generated and this call is unifiable with the head of both clause 22 and clause 30. However, the final program is semi-deterministic because each call with a ground argument is unifiable with the head of at most one non-unit clause (see Section 4).

In order to better understand the program improvements we achieved in the specialized program made out of clauses 21, 22, 28, and 30, we now revisit that derivation with reference to the SLD-tree in Figure 1. Let us consider the graph obtained by identifying the nodes of the SLD-tree which are labelled by identical goals (see Figure 2). Then the set of goals of this graph, except for the root-goal and the empty goal □, can be partitioned into triples of goals of the form $\langle new1(0^{n-k}1), new2(0^{n-k}1), new3(0^{n-k}1)\rangle$, for $k > 0$, such that the goals occurring in the $(k+1)$-st triple are obtained by applying an SLD-resolution step to each goal occurring in the k-th triple. These triples are boxed in Figure 2. The goals in the 'generalized' triple $\langle new1(W), new2(W), new3(W)\rangle$ are the bodies of the clauses 18, 19, and 20 introduced by the definition of the new predicate *new*123, and by finding a recursive definition of this predicate we are able to visit in a breadth-first manner the graph in linear time.

The above example shows one important feature of our technique, that is, its ability of putting together various computations which are performed by the initial program in different branches of the computation tree, so that common repeated subcomputations can be avoided. This feature is also peculiar to the *tupling* strategy for the transformation of functional languages [13], and we will show in the rest of the paper that by extending partial evaluation with tupling-like techniques results in an extremely powerful specialization method.

3 Transformation Rules for Program Specialization

In this section we describe the program transformation rules which we use for program specialization. Similar rules have been already considered in the unfold/fold approach to logic program transformation (see, for instance, [6, 18]). However, their use for program specialization is novel and, more significantly, we propose a strategy which uses these rules and performs program specializations in an automatic way (see Section 5). This strategy allows us to reduce the nondeterminism of the derived program and it may achieve exponential speedups as shown by the previous parsing example.

In what follows we consider definite logic programs extended with inequalities of the form $t_1 \neq t_2$. This extension is required because our case split rule introduces those inequalities. We assume that a logic program is a set of clauses whose bodies are sequences of literals. Each literal is either an atom, that is, a positive literal, or an inequality. We assume that in every program the equality predicate '$=$' is defined by the clause $X = X \leftarrow$.

The declarative semantics of a logic program P is its least Herbrand model $M(P)$, where we have that $t_1 \neq t_2$ holds iff t_1 and t_2 are two different ground terms. Equality may be axiomatized by a complete, decidable theory, called Clark's Equality Theory [11], or CET for short, such that $\forall X_1, \ldots, X_m. t_1 = t_2$ is provable in CET iff t_1 and t_2 are syntactically identical and $\forall X_1, \ldots, X_m. t_1 \neq t_2$ is provable in CET iff t_1 and t_2 are not unifiable.

The operational semantics of our logic programs is based on LDNF-resolution, that is, SLDNF-resolution with the Prolog selection rule which selects the leftmost atom in the goal at hand (see, for instance, [1]). We will also consider the corresponding concept of LDNF-derivations, that is, derivations using LDNF-resolution, and the concept of left-termination, that is, finiteness of all LDNF-derivations.

For the evaluation of inequalities we use negation as failure as follows: for two given, possibly non-ground, terms t_1 and t_2, we have that $t_1 \neq t_2$ succeeds iff the equality $t_1 = t_2$ fails (that is, t_1 and t_2 are not unifiable), and $t_1 \neq t_2$ fails iff $t_1 = t_2$ succeeds with empty computed answer substitution (that is, t_1 and t_2 are identical). In the case where $t_1 = t_2$ succeeds with non-empty computed answer substitution then the evaluation of $t_1 \neq t_2$ flounders. This way of evaluating inequalities is sound w.r.t. the least Herbrand model, but it is not complete.

During program specialization we will feel free to rearrange the order of the clauses and to delete duplicate clauses. However, the rearrangement of the atoms in the body of each clause is not allowed. This is an essential condition for ensuring that our transformation technique preserves left-termination.

Now we present our transformation rules and then we will return on the issue of the semantics which they preserve. We assume that we have constructed from the initial program P_0 the sequence $P_0, \ldots, P_k$ of programs and we construct the next program P_{k+1} in the sequence by applying one of the rules.

R1. *Definition Rule.* We derive the new program P_{k+1} by adding to P_k the following n (≥ 1) clauses:

$$\begin{cases} newp(X_1, \ldots, X_h) \leftarrow Body_1 \\ \ldots \\ newp(X_1, \ldots, X_h) \leftarrow Body_n \end{cases}$$

where: i) *newp* is a new predicate symbol, ii) for $1 \leq i \leq h$, the variable X_i occurs in $Body_j$ for some j such that $1 \leq j \leq n$, and iii) for $1 \leq j \leq n$, the goal $Body_j$ contains one or more atoms with predicate occurring in the initial program P_0.

When specializing programs, during the construction of the sequence $P_0, \ldots, P_k$ of programs, we store in the set Def_k, for $k \geq 0$, all clauses, called *definition clauses*, which have been introduced by the definition rule. Obviously, $Def_0 = \{\}$.

R2. *Unfolding Rule.* Let C be a clause in program 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 $D_1, \ldots, D_n$, with $n \geq 0$, be the clauses of P_k such that for $i = 1, \ldots, n$, A is unifiable with the head of D_i with m.g.u. θ_i. For $i = 1, \ldots, n$, let C_i be the clause $(H \leftarrow F, bd(D_i), G)\theta_i$, where $bd(D_i)$ is the body of D_i. Then by unfolding C w.r.t. A using $D_1, \ldots, D_n$ we replace C by the new clauses $C_1, \ldots, C_n$, thereby deriving program P_{k+1}.

Notice that an application of the unfolding rule to clause C amounts to the deletion of C if $n=0$. Sometimes in the literature this particular case is treated as an extra rule.

R3. *Folding Rule.* Let $C_1, \ldots, C_n$ be clauses in P_k. Let $D_1, \ldots, D_n$ be clauses in Def_k of the form: $newp(X_1, \ldots, X_h) \leftarrow Body_1, \ldots, newp(X_1, \ldots, X_h) \leftarrow Body_n$ and let these clauses be all those in Def_k which have *newp* as head predicate. Suppose that there exist two goals F and G, and a substitution θ such that for $i = 1, \ldots, n$, the following two conditions hold:
1) C_i is a variant of the clause $H \leftarrow F, Body_i\theta, G$, and
2) for every variable X in D_i and not in $\{X_1, \ldots, X_h\}$, we have that: (i) $X\theta$ is a variable which does not occur in (H, F, G), and (ii) $X\theta$ does not occur in $Y\theta$, for any variable Y occurring in $Body_i$ and different from X. By folding $C_1, \ldots, C_n$ using $D_1, \ldots, D_n$ we replace $C_1, \ldots, C_n$ by the single clause $H \leftarrow F, newp(X_1, \ldots, X_h)\theta, G$, thereby deriving program P_{k+1}.

For instance, the clauses C_1: $p(X) \leftarrow q(t(X), Y), r(Y)$ and C_2: $p(X) \leftarrow s(X), r(Y)$ can be folded (via the empty substitution) using the clauses D_1: $a(U, V) \leftarrow q(t(U), V)$ and D_2: $a(U, V) \leftarrow s(U)$, and we replace C_1 and C_2 by the clause C: $p(X) \leftarrow a(X, Y), r(Y)$.

R4. *Clause Subsumption Rule.* We derive P_{k+1} by deleting from P_k a clause which is subsumed by another clause in P_k. (Recall that $(H \leftarrow Body, G)\theta$ is subsumed by $H \leftarrow Body$.)

R5. *Head Generalization Rule.* Let C be a clause of the form: $H\{X/t\} \leftarrow Body$ in P_k, where $\{X/t\}$ is a substitution such that X occurs in H and X does not occur in the term t. We derive the program P_{k+1} by replacing C by the clause $H \leftarrow X = t, Body$.

This rule is a particular case of the rule of 'generalization + equality introduction' considered, for instance, in [15].

R6. *Case Split Rule.* Let C be a clause in P_k of the form: $H \leftarrow Body$. We perform case split of C w.r.t. $\{X/t\}$, thereby deriving P_{k+1}, by replacing C by the following two clauses: $(H \leftarrow Body)\{X/t\}$ and $H \leftarrow X \neq t, Body$, where X is a variable occurring in H and t is any term.

R7. *Simplification Rule.* Let I_1 and I_2 be conjunctions of inequalities such that $\forall X_1, \ldots, X_h. I_1 \leftrightarrow I_2$ holds in CET, where $X_1, \ldots, X_h$ are the variables occurring in $I_1 \leftrightarrow I_2$. We derive P_{k+1} by replacing a clause of the form: $H \leftarrow F, I_1, G$ by $H \leftarrow F, I_2, G$. In particular, the clause $H \leftarrow F, I_1, G$ may be replaced by $H \leftarrow F, G$ if I_2 is *true*, and the clause $H \leftarrow F, I_1, G$ may be deleted if I_2 is *false*.

Thus, by the simplification rule, we may delete $t_1 \neq t_2$ from the body of a clause if t_1 and t_2 are not unifiable, we may delete a clause whose body contains $t_1 \neq t_2$ if t_1 and t_2 are identical, and we may use commutativity and idempotency of conjunctions of inequalities.

A formal proof of the correctness of rules R1–R7 is outside the scope of this paper, which is mainly devoted to the presentation of our automatic transformation strategy for program specialization. However, as we now indicate, in our program transformations we preserve the chosen declarative and operational semantics and this can be shown by extending already known results which ensure the preservation of the least Herbrand model of definite programs.

In order to do so we consider the inequality as a predicate on its own, instead of the negation of the equality predicate, so that our programs can be considered to be definite programs.

In Theorem 1 of [6] it is proved that if some conditions on the folding rule are satisfied, then rules R2, R3, and R6 are *totally correct* w.r.t. the least Herbrand model semantics, that is, $M(P_k) = M(P_0)$, where P_k is obtained from P_0 by applying R2, R3, and R6. In our case, these conditions are implied by the fact that, as stated by the strategy of Section 5, each clause in a set of clauses to which the folding rule is applied, is derived by first unfolding a clause w.r.t. an atom different from an equality and then possibly applying some more transformation rules. This total correctness result can be extended to the case when also the definition rule R1 is considered because we may assume that all definition clauses which are needed, are introduced by performing definition steps at the beginning of the transformation process.

The total correctness w.r.t. the least Herbrand model semantics of the rules R4, R5, and R7 can be proved by reducing them to special cases of the totally correct rules described in [18]. Indeed, clause subsumption is an instance of the clause deletion rule. Moreover, the simplification rule can be viewed as a special case of the goal replacement rule of [18], because equivalences provable in CET which hold between conjunctions of inequalities, are true in the Herbrand models of every definite program. Finally, the case split rule can be viewed as a special case of the goal replacement and unfolding rules as follows. Consider the substitution $\{X/t\}$ where $Y_1, \ldots, Y_h$ are the variables occurring in t. We have that the atom $c(X, Y_1, \ldots, Y_h)$ defined by the two clauses: $c(X, Y_1, \ldots, Y_h) \leftarrow X = t$, and $c(X, Y_1, \ldots, Y_h) \leftarrow X \neq t$, is equivalent to *true* and thus it can be added to the body of any clause by the goal replacement rule of [18]. By adding the atom $c(X, Y_1, \ldots, Y_h)$ to the body of a clause C of the form: $H \leftarrow Body$, and then unfolding the derived clause w.r.t. $c(X, Y_1, \ldots, Y_h)$ and w.r.t. the equality $X = t$ we get the same clauses which can be derived by case split of C w.r.t. $\{X/t\}$.

Now, the argument we have presented does not prove that any combination of the rules R1–R7 is *totally correct* w.r.t. the least Herbrand model semantics, because the proof of total correctness of R2, R3, and R6 in [6] makes use of an invariant which is preserved by these rules (if some conditions on the folding rule are satisfied). This invariant is based on Lemma 9 of [6] where it is proved that the so called *weight consistency* is preserved. Thus, in order to prove our total correctness result concerning rules R1–R7, we need to prove that also R1, R4, R5, and R7 preserve the invariant. This can be done in a straightforward way, but we omit this proof.

Moreover, we claim that by applying rules R1–R7 according to our strategy for program specialization given in Section 5, also left-termination is preserved, in the sense that for any goal G in a set of goals of interest (see Section 4) we have that if $P_0 \cup Def_k$ left-terminates for the goal G then P_k left-terminates for the goal G.

By using Bossi and Cocco's result (see Theorem 21 in [1]) we can prove the preservation of left-termination of definite programs, but only in the case where the following four kinds of rules are used: i) definition, with the restriction that the new predicate is defined by one clause only; ii) unfolding; iii) folding, with the restriction that any clause to which the folding rule is applied has been derived by performing at least an unfolding step of a clause w.r.t. the leftmost atom different from an equality; and iv) any other *non-increasing* and *complete* transformation rule applied to clauses which are not in P_0 (see Definitions 5 and 7 in [1]). In our transformation strategy of Section 5 the condition on the folding rule is satisfied, because at the beginning of each iteration of the strategy we first unfold the clauses at hand w.r.t. all equalities in the body and then w.r.t. the leftmost atom which is not an equality.

Now, in order to prove the preservation of left-termination for the whole set of rules R1–R7, we may adapt the proofs in [1] by using the concept of LDNF-derivation instead of LD-derivation, and this is not a difficult task because we use negation as failure only for evaluating inequalities. We may also adapt the concept of non-increasingness, which refers to the length of the LD-derivations. Indeed, if we do not count the steps of evaluation of the inequalities which are introduced by case split, then the length of the LDNF-derivations is not increased by applying any of the rules R1–R7 according to our strategy. The reader may notice that Theorem 21 in [1] requires also the completeness property, that is, the fact that the transformations preserve the computed answer substitutions. Here we do not need this property because we are interested in preserving the the least Herbrand model only. And indeed, by using our rules, operational observables, such as the order and the multiplicity of the computed answer substitutions, may not be preserved. This is due to the use of clause subsumption and clause reordering, this last rule being implicit in the concept of logic program we consider.

4 Program Transformation based on Modes

Our program transformation rules may be used for increasing the amount of determinism in logic programs. The concept of determinism we use, called *semi-determinism*, is based on the concept of program *modes*. Program modes provide an abstract interpretation framework which suggests guidelines for the application of the transformation rules. These guidelines are incorporated into our strategy given in Section 5.

The *mode* of a predicate p of arity h is denoted by an expression of the form $p(m_1, \ldots, m_h)$, where m_i, for $i = 1, \ldots, h$, is either g (short for ground) or u (short for unknown). In intuitive terms, by $m_i = g$ we mean that at runtime when p is called, the i-th argument of p is ground, while by $m_i = u$ we mean that we do not know whether or not the i-th argument of p is ground when p is called.

The mode $p(m_1, \ldots, m_h)$ is the set of all literals of the form $p(t_1, \ldots, t_h)$ such that, for $i = 1, \ldots, h$, if $m_i = g$ then t_i is a ground term, and if $m_i = u$ then t_i is any term. For $i = 1, \ldots, h$, the i-th argument position of the predicate p is said to be an *input position* for p iff $m_i = g$ in the mode of p. We assume that the mode of the predicate '$=$' is $u = u$ and the mode of the predicate '$\neq$' is $g \neq u$. The mode of a program P is the union of the modes of the distinct predicate symbols (including those for equality and inequality) in P.

Definition 1. A mode M is *sound* w.r.t. a given program P and set I of goals, called *goals of interest*, iff each literal occurring at the leftmost position in a goal of an LDNF-derivation which uses P and starts from a goal in I, belongs to M.

In most cases the soundness of a mode can be automatically verified by well-known abstract interpretation methods (see, for instance [3]).

Now we introduce the concept of semi-determinism for our logic programs, that is, definite programs together with inequalities.

Definition 2. A logic program P is *semi-deterministic* w.r.t. a set I of goals of interest iff for each goal G in I, and for each literal A different from an inequality occurring at the leftmost position in a goal derived from G by LDNF-resolution, there exists *at most one* non-unit clause C in P such that the head of C is unifiable with A via an m.g.u. θ and the instances via θ of all inequalities in the body of C succeed.

Notice that a semi-deterministic program may allow for multiple answers for a given goal. This is the main difference with the programs with *functional predicates* defined in [4]. For instance, the following program is semi-deterministic w.r.t. the set of goals belonging to the mode $p(g, u) \cup q(u)$ (but p is not a functional predicate w.r.t. its first argument): $p(0, 1) \leftarrow$, $p(0, Y) \leftarrow q(Y)$, $q(1) \leftarrow$, $q(2) \leftarrow$.

We now give a simple condition which ensures semi-determinism. It is based on the following concept of *mutually exclusive* clauses.

Definition 3. Let us consider the following two clauses:

C_1: $p(\ldots, t_1, \ldots) \leftarrow Ineqs_1, B_1$
C_2: $p(\ldots, t_2, \ldots) \leftarrow Ineqs_2, B_2$

where i) t_1 and t_2 are terms occurring at the *same* argument position which is the only input position w.r.t. a given mode M, ii) $Ineqs_1$ and $Ineqs_2$ are sequences of inequalities, and iii) B_1 and B_2 are sequences of atoms. C_1 and C_2 are said to be *mutually exclusive* w.r.t. mode M iff *either* t_1 and t_2 are not unifiable *or* they are unifiable via an m.g.u. θ and at least one inequality in $\{Ineqs_1\theta, Ineqs_2\theta\}$ fails.

In order to extend Definition 3 to the case where more than one input position is considered, it is enough to consider h arguments together as a single argument which is an h-tuple.

Proposition 4. Let P be a program, M a mode, and I a set of goals such that $M \supseteq I$. If M is sound w.r.t. P and I and the non-unit clauses in P are pairwise mutually exclusive w.r.t. M, then P is semi-deterministic w.r.t. I.

One important feature of our transformation strategy is that, if it terminates, it derives semi-deterministic programs. In order to verify this property we use the above proposition and the following Theorem 5 which states that when we apply our transformation rules we preserve the soundness of the modes and we can define sound modes for the new predicates introduced by definition rule.

Theorem 5. Let P_0 be a program, I be a set of goals of interest, and $M_0 \supseteq I$ be a mode which is sound w.r.t. P_0 and I. Let $P_0, \ldots, P_s$ be a sequence of programs derived from P_0 by applying the transformation rules R1–R7. Let $M_0, \ldots, M_s$ be modes of $P_0, \ldots, P_s$, respectively, such that, for $k = 0, \ldots, s$, $M_k = M_0 \cup NewM_k$, where $NewM_k$ is the union of the modes of the new predicates introduced during the construction of the sequence $P_0, \ldots, P_k$. Suppose that, for $k = 0, \ldots, s-1$, the following conditions are satisfied:
1) *if* P_{k+1} is derived from P_k by case split of a clause C w.r.t. $\{X/t\}$, *then* X occurs at an input position in the head of C according to mode M_k;
2) *if* P_{k+1} is derived from P_k by applying the folding rule and replacing the clauses $H \leftarrow F, Body_i\theta, G$, for $i = 1, \ldots, n$, by the new clause $H \leftarrow F, newp(\ldots)\theta, G$ using the definition clauses $newp(\ldots) \leftarrow Body_i$, for $i = 1, \ldots, n$, *then* the following holds:
Property (α): each variable occurring in $newp(\ldots)\theta$ at an input position according to the mode M_k, also occurs in one of the atoms $H, F, A_1\theta, \ldots, A_n\theta$ at an input position, where $A_1, \ldots, A_n$ are the leftmost atoms of $Body_1, \ldots, Body_n$, respectively.

Then M_s is sound w.r.t. P_s and I.

Proof. The proof proceeds by induction on the sequence $P_0, \ldots, P_s$. We assume that the mode M_k, with $0 \leq k \leq s-1$, is sound w.r.t. the program P_k and I, and we prove that the mode M_{k+1} is sound w.r.t. P_{k+1} and I. We will free to omit the reference to the set I of goals of interest.

If P_{k+1} is obtained by applying the definition rule, we have to provide a mode for the newly introduced predicate, say *newp*. Since no LDNF-derivation relative to P_{k+1} which starts from a goal in I may contain calls to *newp*, then for each choice of the mode N of *newp* we have that $M_k \cup N$ is sound w.r.t. P_{k+1} and I.

When applying rules R2–R7 we have that $M_{k+1} = M_k$, because no new predicate is introduced, and thus, we only have to show that the soundness of M_k is preserved.

If P_{k+1} is obtained by applying the unfolding rule, then the soundness of the mode M_k w.r.t. P_{k+1} derives from the soundness of the mode M_k w.r.t. P_k, because this rule preserves the order of the atoms in the bodies of the clauses and the instantiation of the clauses does not decrease. (Recall that the order of the clauses is immaterial for establishing the soundness of the modes.)

Let us now examine the case where P_{k+1} is obtained by applying the folding rule as stated in Condition 2. This folding step may affect the soundness of the mode of *newp* because LDNF-derivations relative to P_{k+1} may contain calls to *newp*, which do not occur in those relative to P_k. However, if Property (α) holds, the soundness of M_k w.r.t. P_{k+1} derives from the soundness of M_k w.r.t. P_k, because the groundness of the arguments in calls of *newp* which correspond to input positions according to M_k, is implied by the modes of the predicates occurring in $H, F, A_i\theta$, where A_i is the leftmost atom in $Body_i$, for $i = 1, \ldots, n$.

If P_{k+1} is obtained by applying the clause subsumption rule, the soundness of M_k w.r.t. P_{k+1} derives from that of M_k w.r.t. P_k, because all LDNF-derivations relative to P_{k+1} are also LDNF-derivations relative to P_k. The same property of LDNF-derivations holds, except for the absence of some inequalities, when P_{k+1} is obtained by applying the simplification rule.

If P_{k+1} is obtained by applying the case split rule to a clause C, the soundness of M_k w.r.t. P_{k+1} derives from that of M_k w.r.t. P_k, because the literals occurring in the LDNF-derivations relative to P_{k+1} cannot be less instantiated than those in the LDNF-derivations relative to P_k. They may also contain literals of the form $(X \neq t)\theta$, where X is a variable, t is a term, and θ a substitution. Since we assume that the mode of '$\neq$' is $g \neq u$ and by hypothesis X occurs at an input position in the head of the clause C, then M_k is sound also w.r.t. P_{k+1}.

Finally, when P_{k+1} is obtained by applying the head generalization rule, the soundness of M_k w.r.t. P_{k+1} is a consequence of the fact that M_k is sound w.r.t. P_k, because the equalities are inserted at the leftmost positions in the bodies of the clauses, and the mode of the equality predicate is $u = u$. □

Since the choice of the mode for a new predicate, say *newp*, introduced by the definition rule is arbitrary, one trivial way of satisfying Property (α) is to assume that for *newp* there are no input positions, that is, $newp(u, \ldots, u)$. However, in order to retain in the mode of *newp* some information for guiding the application of the various transformation rules, in our strategy given in Section 5.3 we will indicate how one may choose for *newp* a mode which is different from $newp(u, \ldots, u)$ and satisfies Property (α).

5 A Transformation Strategy for Reducing Nondeterminism

In this section, we present a strategy for guiding the application of the transformation rules presented in Section 3 so to derive specialized programs and reduce nondeterminism. In particular we will be able to prove that the derived programs are semi-deterministic, but unfortunately, we have no formal proof of the efficiency improvements obtained by using our transformation strategy. However, in this section we show through an example that this strategy is able to produce very efficient specialized programs.

Given an initial definite program P with mode M, and an *atomic* goal $p(t_1, \ldots, t_r)$ w.r.t. which we want to specialize P, we introduce by the definition rule the specializing clause C: $sp\_p(X_1, \ldots, X_h) \leftarrow p(t_1, \ldots, t_r)$, where $X_1, \ldots, X_h$ are the distinct variables occurring in $p(t_1, \ldots, t_r)$. We also define the mode $sp\_p(m_1, \ldots, m_h)$ of the predicate sp_p by stipulating that: for any $j = 1, \ldots, h,\ m_j = g$ iff X_j occurs at an input position in $p(t_1, \ldots, t_r)$ according to the mode M.

We assume that the mode M for P is sound w.r.t. the set of goals of interest consisting of the instances of $p(t_1, \ldots, t_r)$ which belong to M, and as a consequence, also the mode $M \cup sp\_p(m_1, \ldots, m_h)$ for program $P \cup \{C\}$ is sound w.r.t. the set $sp\_p(m_1, \ldots, m_h)$ of goals of interest. Obviously, for any R contained in $sp\_p(m_1, \ldots, m_h)$, the mode $M \cup R$ is sound w.r.t. the program P and the set R of goals of interest, but a restriction of the mode of the specialized predicate sp_p corresponds to an undesirable restriction of the set of the goals of interest.

The transformation strategy we now define, is described as an iterative program which at the i-th iteration, for $i \geq 0$, transforms a program of the form: $P \cup T_i \cup N_i$ into a program of the form: $P \cup T_{i+1} \cup N_{i+1}$. This strategy takes as parameters some subsidiary strategies which may be modified according to the particular specialization needs. We will present these subsidiary strategies in the following subsections.

At the end of the i-th iteration, when the program $P \cup T_{i+1} \cup N_{i+1}$ has been derived, all definition clauses introduced so far are stored in the set Def_{i+1}, and the mode of $P \cup T_{i+1} \cup N_{i+1}$, which is a superset of the mode M_i of $P \cup T_i \cup N_i$, is the set M_{i+1}.

The Transformation Strategy

Initialization: $i = 0$;

$T_0 = \{\}$; $N_0 = Def_0 = \{C\}$; $M_0 = M \cup sp\_p(m_1, \ldots, m_h)$;

While $N_i \neq \{\}$ *do*:

1. *Unfolding Steps*: For each clause D in N_i we first unfold D w.r.t. all equalities, if any, in its body. (Equalities could have been introduced in the previous iteration by the head generalization step, as we will see.) Then we unfold the leftmost atom in the body of the resulting clause thereby deriving a set of clauses. This set is then transformed according to the subsidiary unfolding strategy given in Section 5.1. Thus, we derive a new program $P \cup T_i \cup E_i$.

2. *Case Split, Simplification, Subsumption, and Head Generalization Steps*: By case split, simplification, subsumption, and head generalization steps, from E_i we derive a set F_i of clauses with the following property: the set of non-unit clauses of F_i are partitioned into *packets*, that is, disjoint subsets of clauses such that: (i) any two clauses in different packets are mutually exclusive w.r.t. mode M_i, and (ii) any two clauses in the same packet have equal heads and the same set of inequalities introduced by case split (up to a variable renaming). The detailed strategy used for directing the case split, simplification, subsumption, and head generalization steps is given in Section 5.2.

3. *Definition and Folding Steps*: Initialize Def_{i+1} to Def_i. Then for each packet of clauses in F_i we perform the following transformation steps. Let the packet under consideration be of the form:

$$\begin{cases} H \leftarrow \; Ineqs, \; Body_1 \\ \quad \ldots \\ H \leftarrow \; Ineqs, \; Body_n \end{cases}$$

where we assume that *Ineqs* are (possibly empty) sequences of inequalities introduced by case split (recall that the heads and the sets of inequalities in each clause of a packet are identical modulo renaming). By applying zero or more times the definition rule, we introduce zero or more new predicates with their modes, by adding to Def_{i+1} a non-empty set of definition clauses for each of these new predicates. These definition steps are such that:

(a) we can then perform one or more folding steps using clauses in Def_{i+1} so that the packet under consideration is replaced by a single clause of the form:

$$H \leftarrow Ineqs, newq_1(\ldots)\theta_1, \ldots, newq_r(\ldots)\theta_r$$

where $newq_1, \ldots, newq_r$ are predicates defined by clauses in Def_{i+1}, and

(b) Property (α) of Theorem 5 in Section 4 holds for the chosen modes of $newq_1, \ldots, newq_r$.

These definition and folding steps will be performed according to the subsidiary strategy given in Section 5.3. T_{i+1} is obtained by adding to T_i all clauses (one for each packet) derived by folding, and also every unit clause in E_i. N_{i+1} is the set of clauses occurring in the final value of Def_{i+1} and not in Def_i. M_{i+1} is obtained by adding to M_i the modes for the head predicates in N_{i+1}.

Increment: $i := i + 1$

end_do □

Suppose that the strategy terminates at step s, then the program derived by our transformation strategy is $P \cup T_s$, because $N_s = \{\}$. By the correctness of the transformation rules, for $0 \leq i \leq s-1$, $P \cup T_i \cup N_i$ is equivalent to $P \cup T_{i+1} \cup N_{i+1}$ w.r.t. the semantics of logic programs as we have indicated at the end of Section 3, and thus, $P \cup T_s$ is equivalent to $P \cup N_0$. Moreover, since no predicate occurs both in P and in T_s, we can delete P because the goals of interest depend on predicates in T_s only.

To show that T_s is a semi-deterministic program w.r.t. the set $sp\_p(m_1, \ldots, m_h)$ of the goals of interest, we observe that at each iteration:

1) the conditions on the applications of the case split and folding rules of Theorem 5 are satisfied, and

2) the non-unit clauses of T_i are mutually exclusive w.r.t. mode M_i.

By Point 1) and Theorem 5, the mode M_s is sound w.r.t. T_s and the set $sp\_p(m_1, \ldots, m_h)$ of goals of interest. By Point 2), Proposition 4, and the soundness of M_s, we have that T_s is semi-deterministic w.r.t. $sp\_p(m_1, \ldots, m_h)$.

In order to complete the description of our transformation strategy, we now indicate the subsidiary strategies for applying each transformation rule. We will see these subsidiary strategies in action in the example of Section 5.4.

5.1 Unfolding Strategy

We transform as long as possible the current set of clauses by unfolding any clause w.r.t. the leftmost atom in its body which has a non-variable term occurring in at least one input position (according to the given program mode). Thus, we stop the unfolding strategy when all atoms have variables at their input positions.

The aim of this unfolding strategy is to 'consume' the instantiation produced by previous unfolding steps (recall that in any case we perform an unfolding step w.r.t. the leftmost atom of each clause in N_i).

This strategy differs from usual unfolding strategies for partial evaluation of logic programs because mode information is used. We have performed several experiments with this strategy and we have found it effective on many examples. Unfortunately, our strategy may not terminate. To address this problem we could take advantage of some techniques for finite unfolding (see [5] for this issue).

5.2 Strategy for Case Split, Simplification, Subsumption, and Head Generalization

Given a set E_i of clauses, for every subset, say $E_i(p)$, of all non-unit clauses of E_i whose head predicate is p, we transform $E_i(p)$ by performing as long as possible the following case split, simplification, and subsumption steps.

For simplicity we assume that p is a binary predicate and only its first argument position is an input position w.r.t. mode M_i. For each pair of clauses, say C_1: $p(t_1, u_1) \leftarrow Ineqs_1, Body_1$ and C_2: $p(t_2, u_2) \leftarrow Ineqs_2, Body_2$, such that: i) $\langle t_1, Ineqs_1\rangle$ is not a variant of $\langle t_2, Ineqs_2\rangle$ and ii) t_1 is unifiable with t_2 via an m.g.u. θ containing a binding X/r such that $t_1\{X/r\}$ is not a variant of t_1, we replace in $E_i(p)$ clause C_1 by the two clauses $(p(t_1, u_1) \leftarrow Ineqs_1, Body_1)\{X/r\}$ and $p(t_1, u_1) \leftarrow X \neq r, Ineqs_1, Body_1$. These case split steps are interleaved with applications of the subsumption and simplification rules, whenever possible.

At the end of this process the resulting set of clauses can be partitioned into packets such that: (i) any two clauses in different packets are mutually exclusive w.r.t. mode M_i, and (ii) any two clauses in the same packet have equal arguments *at the input positions* and the same set of inequalities introduced by case split (modulo variable renaming). We now apply head generalization so to replace all non-input arguments in the heads of the same packet by their least common generalization. Finally, we move all inequalities to leftmost positions. Notice that this last step preserves the soundness of the mode M_i because the mode of $\neq$ is '$g \neq u$' and, by the conditions on how to apply the case split rule, the left hand side of each inequality occurs at an input position in the head of the clause.

Thus, we have derived the desired packets of clauses from the non-unit clauses of E_i.

5.3 Definition and Folding Strategy

Several strategies for performing the required definition and folding steps are possible. We now present the simple strategy which we have used in the specialization presented in Section 5.4. This strategy does not guarantee the termination of the specialization process. In order to ensure termination one may adapt generalization techniques used in partial evaluation (see the Partial Evaluation Strategy in Section 1 and [5, 8, 10]). Let us consider the following packet:

$$\begin{cases} H \leftarrow Ineqs,\ Body_1 \\ \quad \dots \\ H \leftarrow Ineqs,\ Body_n \end{cases}$$

and let us suppose that the set Def_{i+1} of the available definition clauses contains a subset of clauses of the form:

$$\begin{cases} newq(X_1, \dots, X_h) \leftarrow Body_1 \\ \quad \dots \\ newq(X_1, \dots, X_h) \leftarrow Body_n \end{cases}$$

such that (i) they are all clauses in Def_{i+1} with head predicate $newq$, (ii) $X_1, \dots, X_h$ include every variable which occurs in one of the goals $Body_1, \dots, Body_n$ and also occurs in one of the goals $H, Ineqs$ (this property is needed for the correctness of folding, see Section 3), and (iii) Property (α) of Theorem 5 holds for the mode of $newq$.

Then we fold the given packet and we get:

$H \leftarrow Ineqs,\ newq(X_1, \dots, X_h)$.

Otherwise, if in Def_{i+1} there is no such a set of definition clauses, we add to Def_{i+1} the following clauses for a new predicate $newr$:

$$\begin{cases} newr(X_1, \dots, X_v) \leftarrow Body_1 \\ \quad \dots \\ newr(X_1, \dots, X_v) \leftarrow Body_n \end{cases}$$

where $X_1, \dots, X_v$ are all variables which occur in one of the goals $Body_1, \dots, Body_n$ and also occur in one of the goals $H, Ineqs$. We stipulate that the mode $newr(m_1, \dots, m_v)$ is such that, for $i = 1, \dots, v$, $m_i = g$ iff X_i occurs in H at an input position, and thus, Property (α) of Theorem 5 holds for the mode of $newr$. We then fold the packet under consideration and we get:

$H \leftarrow Ineqs, newr(X_1, \dots, X_v)$.

5.4 An Application of the Transformation Strategy

In this example we see in action our transformation strategy for the specialization of a multi-pattern matching program and we derive a very efficient semi-deterministic program. A similar specialization could not be performed by usual partial evaluation techniques without a prior transformation into 'failure continuation passing style' form [17]. We are given the initial program:

1. $mmatch([P|Ps], S, N) \leftarrow match(P, S, N)$
2. $mmatch([P|Ps], S, N) \leftarrow mmatch(Ps, S, N)$
3. $match(P, S, N) \leftarrow app(Y, R, S),$
   $app(L, P, Y),\ len(L, N)$
4. $len([\,], 0) \leftarrow$
5. $len([H|T], s(N)) \leftarrow len(T, N)$
6. $app([\,], L, L) \leftarrow$
7. $app([A|L1], L2, [A|L3]) \leftarrow app(L1, L2, L3)$

The predicate $mmatch(Ps, S, N)$ holds iff there exists a pattern in the list Ps of patterns which occurs in the string S at position N. We assume that the mode M of the program is given by the following modes of the various predicates: $mmatch(g, g, u)$, $match(g, g, u)$, $app(u, u, g)$, $len(g, u)$. We do not provide here the proof of the soundness of M.

The reader should notice that the derivation we will perform is more challenging than the ones usually presented in the literature (see, for instance, [7]) because it is a multi-pattern specialization (not a single pattern) and also because in the initial program the occurrence of a pattern in the string is specified via list concatenation using the append predicate *app*.

We want to specialize this multi-pattern matching program w.r.t. the list $[[a,a,a],\ [a,a,b]]$ of patterns. Thus, we introduce the specializing clause:

8. $sp\_mmatch(S,N) \leftarrow$
$\qquad mmatch([[a,a,a],[a,a,b]],S,N)$

The mode of the new predicate is $sp\_mmatch(g,u)$ because S occurs at an input position of *mmatch*, and N does not. Thus, our transformation strategy starts off with the following initialization: $T_0 = \{\}$, $N_0 = Def_0 = \{8\}$, and $M_0 = M \cup sp\_mmatch(g,u)$.

We will not show here the complete derivation, but in order to exemplify all subsidiary strategies of Sections 5.1, 5.2, and 5.3, we will present the second iteration of our strategy. This iteration starts off with $T_1 = \{9\}$, $N_1 = \{10, 11\}$, $Def_1 = \{8, 10, 11\}$, and $M_1 = M_0 \cup new1(g,u)$, where the clauses 9, 10, and 11 listed below have been derived during the first iteration.

9. $sp\_mmatch(S,N) \leftarrow new1(S,N)$
10. $new1(S,N) \leftarrow app(Y,R,S), app(L,[a,a,a],Y),$
$\qquad len(L,N)$
11. $new1(S,N) \leftarrow app(Y,R,S), app(L,[a,a,b],Y),$
$\qquad len(L,N)$

Unfolding Steps: We first unfold each of the clauses 10 and 11 in N_1 w.r.t. the leftmost atom in its body, and we get

12. $new1(S,N) \leftarrow \underline{app(L,[a,a,a],[\,])}, len(L,N)$
13. $new1([C|S],N) \leftarrow app(Y,R,S),$
$\qquad \underline{app(L,[a,a,a],[C|Y])}, len(L,N)$
14. $new1(S,N) \leftarrow \underline{app(L,[a,a,b],[\,])}, len(L,N)$
15. $new1([C|S],N) \leftarrow app(Y,R,S),$
$\qquad \underline{app(L,[a,a,b],[C|Y])}, len(L,N)$

Now we unfold clauses 12, 13, 14, and 15 according to the subsidiary strategy described in Section 5.1, that is, we unfold each of these clauses w.r.t. the leftmost atom of its body which has a non variable term at an input position (see the underlined atoms). The unfolding of clauses 12 and 14 amounts to their deletion, because the selected atoms do not unify with any head in the current program. The unfolding of clauses 13 and 15 yields some new clauses that are further unfolded. After some unfolding steps, we are left with the clauses 16, 17, 18, and 19 listed below, and we stop unfolding because all atoms in their bodies have variables at input positions. Thus we have $E_1 = \{16, 17, 18, 19\}$.

16. $new1([a|S],0) \leftarrow app([a,a],R,S)$
17. $new1([C|S],s(N)) \leftarrow app(Y,R,S),$
$\qquad app(L,[a,a,a],Y), len(L,N)$
18. $new1([a|S],0) \leftarrow app([a,b],R,S)$
19. $new1([C|S],s(N)) \leftarrow app(Y,R,S),$
$\qquad app(L,[a,a,b],Y), len(L,N)$

Case Split, Simplification, Subsumption, and Head Generalization Steps: We apply case split to clause 17 w.r.t. to the substitution $\{C/a\}$, because the input argument of the head of this clause is unifiable with the input argument of the head of clause 16 via the m.g.u. $\{C/a\}$. We derive the following clause:

20. $new1([a|S],s(N)) \leftarrow app(Y,R,S),$
$\qquad app(L,[a,a,a],Y), len(L,N)$
21. $new1([C|S],s(N)) \leftarrow C \neq a, app(Y,R,S),$
$\qquad app(L,[a,a,a],Y), len(L,N)$

Similarly, by applying case split to clause 19 we derive:

22. $new1([a|S],s(N)) \leftarrow app(Y,R,S),$
$\qquad app(L,[a,a,b],Y), len(L,N)$
23. $new1([C|S],s(N)) \leftarrow C \neq a, app(Y,R,S),$
$\qquad app(L,[a,a,b],Y), len(L,N)$

Now, the set of clauses derived so far from E_1 can be partitioned in two packets: the first one is made out of the clauses 16, 18, 20, and 22 whose input argument of the head is of the form $[a|S]$, and the second one is made out of the clauses 21 and 23 whose input argument of the head is of the form $[C|S]$ with $C \neq a$.

We apply head generalization to clauses 16, 18, 20 and 22, and we replace the second arguments in their heads by the least common generalization of those arguments, that is, a variable. We get the packet:

24. $new1([a|S],M) \leftarrow M=0, app([a,a],R,S)$
25. $new1([a|S],M) \leftarrow M=0, app([a,b],R,S)$
26. $new1([a|S],M) \leftarrow M=s(N), app(Y,R,S),$
$\qquad app(L,[a,a,a],Y), len(L,N)$
27. $new1([a|S],M) \leftarrow M=s(N), app(Y,R,S),$
$\qquad app(L,[a,a,b],Y), len(L,N)$

For the packet made out of the clauses 21 and 23 whose heads are variants of each other, no head generalization step is performed. Thus, we have derived the set of clauses F_1 which is the union of the packet $\{24, 25, 26, 27\}$ and the packet $\{21, 23\}$.

Definition and Folding Steps: Initially $Def_2 = Def_1$. In order to fold the packet of clauses $\{24, 25, 26, 27\}$, we introduce the new predicate *new*2 as follows:

28. $new2(S,M) \leftarrow M=0, app([a,a],R,S)$
29. $new2(S,M) \leftarrow M=0, app([a,b],R,S)$
30. $new2(S,M) \leftarrow M=s(N), app(Y,R,S),$
$\qquad app(L,[a,a,a],Y), len(L,N)$
31. $new2(S,M) \leftarrow M=s(N), app(Y,R,S),$
$\qquad app(L,[a,a,b],Y), len(L,N)$

Clauses 28, 29, 30, and 31 are added to Def_2 and they are used for folding clauses 24, 25, 26, and 27. We get the clause:

32. $new1([a|S], M) \leftarrow new2(S, M)$

The mode of $new2$ is $new2(g, u)$ because the variable S occurs at an input position of $new1$ in the head of each clause of the corresponding packet. We also fold the packet $\{21, 23\}$ by using clauses 10 and 11 which were already in Def_1, and we get:

33. $new1([C|S], s(N)) \leftarrow C \neq a, new1(S, N)$

These folding steps conclude the second iteration with $T_2 = \{9, 32, 33\}$, $N_2 = \{28, 29, 30, 31\}$, $Def_2 = Def_1 \cup N_2$, and $M_2 = M_1 \cup new2(g, u)$.

At the sixth iteration, we will be able to perform all required folding steps without introducing new definition clauses and the transformation strategy stops. Our final program is the following:

$sp\_mmatch(S, N) \leftarrow new1(S, N)$
$new1([a|S], M) \leftarrow new2(S, M)$
$new1([C|S], s(N)) \leftarrow C \neq a, new1(S, N)$
$new2([a|S], M) \leftarrow new3(S, M)$
$new2([C|S], s(s(N))) \leftarrow C \neq a, new1(S, N)$
$new3([a|S], M) \leftarrow new4(S, M)$
$new3([b|S], M) \leftarrow new5(S, M)$
$new3([C|S], s(s(s(N)))) \leftarrow C \neq a, C \neq b, new1(S, N)$
$new4(S, 0) \leftarrow$
$new4([a|S], s(N)) \leftarrow new4(S, N)$
$new4([b|S], s(N)) \leftarrow new5(S, N)$
$new4([C|S], s(s(s(s(N))))) \leftarrow C \neq a, C \neq b, new1(S, N)$
$new5(S, 0) \leftarrow$
$new5([a|S], s(s(s(N)))) \leftarrow new2(S, N)$
$new5([C|S], s(s(s(s(N))))) \leftarrow C \neq a, new1(S, N)$

This final program is semi-deterministic and it is about 10 times faster than the initial one for computing all positions where one of the patterns $[a, a, a]$ or $[a, a, b]$ occurs in a random string made out of 100 symbols. The performance of the specialized program may be further improved by removing all inequalities in favour of cuts.

6 Concluding Remarks and Related Work

We have proposed a specialization technique for logic programs based on an automatic strategy which makes use of the following transformation rules: definition, unfolding, folding, subsumption, head generalization, case split, and simplification of inequalities. We have also shown that our strategy may reduce the amount of nondeterminism in the specialized programs, thereby achieving exponential gains in time complexity. Thus, it greatly improves over usual specialization techniques based on partial evaluation.

To achieve these results, we allow new predicates to be introduced by *one or more* non-recursive definition clauses whose bodies may contain *more than one* atom. We also allow folding steps using these definition clauses (see also [6] for a similar transformation rule). Thus, by a folding step, a program where a predicate, say p, occurs in several clause heads can be transformed into an equivalent program where the predicate p occurs in one clause only, thereby reducing nondeterminism.

Our extended definition and folding rules allow us to specialize programs w.r.t. *disjunctions of conjunctions of goals* and they generalize the one proposed for partial evaluation in [9], where only conjunctions of atoms are taken into consideration. The techniques presented in [9] recast in the style of Lloyd and Shepherdson's [12] those for the elimination of intermediate data structures [15], and these techniques neither can combine partial evaluations of alternative branches of the SLDNF-trees, nor can introduce inequalities.

Our definition and folding rules also allow us to factorize common computations in different branches of the SLDNF-trees. In a sense this factorization provides an extension of the basic supercompilation techniques, where the program improvements are achieved by taking into consideration single computation paths only [7, 19].

The use of the clause subsumption rule is motivated by the desire of increasing efficiency by avoiding redundant computations. Head generalizations are used for making equal the heads of several clauses and thus they allow us to perform folding steps.

The case-split rule is the crucial rule for the derivation of semi-deterministic programs because it replaces a clause, say C, by several clauses which correspond to exhaustive and mutually exclusive instantiations of the head of C. To get exhaustiveness and mutual exclusion, we have allowed for the introduction of inequalities. To increase program efficiency in a Prolog-like environment, these inequalities may then be removed in favour of cuts.

Our transformation strategy makes use of mode information for directing the various transformation steps, and in particular the unfolding and case split steps. We assume that the initial program is given together with a sound mode which can be used throughout the entire specialization process because it is preserved by our transformation rules. The preservation of mode soundness is used to show that if our strategy terminates, then we derive semi-deterministic programs. Although semi-determinism is not a guarantee for efficiency improvement, it is often the case that efficiency is increased when nondeterminism is reduced.

The transformation strategy we propose may not terminate. This may be due both to the unfolding strategy, which allows for infinitely many steps, and to the definition and folding strategy, which allows for the introduction of infinitely many new predicates. We believe that the finiteness of our unfolding strategy can be guaranteed by applying techniques already developed for partial evaluation, and in particular, generalization methods for partial evaluation (see, for instance, [5, 10]) can be adapted to our framework for avoiding the definition of an infinite number of new predicates. Notice, however, that this adaptation may not be straightforward because our definition and folding rules are more complex than the ones used in partial evaluation.

We have implemented our proposed strategy and we tested it by performing several program specializations. Our strategy is able to pass the so called KMP test, that is, we were able to fully automatically derive a program which behaves like Knuth-Morris-Pratt algorithm from a general pattern matcher and a given pattern. This was done also in the case of a multi-matcher which uses a naive procedure for testing the occurrence of a pattern in a string. Also other examples, such as the derivation of a deterministic finite automaton by specializing a parser for regular expressions, were successfully worked out. In all these examples the improvement over already known partial evaluation techniques consists in the fact that we start from naive, nondeterministic initial programs, while the corresponding derivations by partial evaluation use initial programs which already incorporate some ingenuity.

With respect to a preliminary version of this paper [14] we made several improvements and, in particular, we used mode information for increasing the generality of our technique and making it easily mechanizable.

Acknowledgments

We would like to thank D. De Schreye, J. Gallagher, R. Glück, N. D. Jones, M. Leuschel, B. Martens, and M. H. Sørensen for stimulating discussions about partial evaluation and logic program specialization. We also acknowledge very useful comments by anonymous referees. This work has been partially supported by the EC under the HCM Project 'Logic Program Synthesis and Transformation', the INTAS Project 93-1702, MURST 40% (Italy), and Progetto Coordinato 'Programmazione Logica' of the CNR (Italy).

References

[1] A. Bossi, N. Cocco. Preserving universal termination through unfold/fold. *Proc. ALP '94*, LNCS 850, pp. 269–286. Springer Verlag, 1994.

[2] R.M. Burstall, J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[3] S.K. Debray, D.S. Warren. Automatic mode inference for logic programs. *J. Logic Programming*, 5:207–229, 1988.

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

[5] J.P. Gallagher. Tutorial on specialization of logic programs. *Proc. PEPM '93, Copenhagen, Denmark*, pp. 88–98. ACM Press, 1993.

[6] M. Gergatsoulis, M. Katzouraki. Unfold/fold transformations for definite clause programs. *Proc. PLILP '94*, LNCS 844, pp. 340–354. Springer Verlag, 1994.

[7] R. Glück, A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. *Proc. WSA '93, Padova, Italy*, LNCS 724, 112–123. Springer Verlag, 1993.

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

[9] M. Leuschel, D. De Schreye, A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. *Proc. JICSLP '96, Bonn, Germany*, pp. 319–332. MIT Press, 1996.

[10] M. Leuschel, B. Martens. Global control for partial deduction through characteristic atoms and global trees. Report CW 220, K.U. Leuven, Belgium, 1995.

[11] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. 2nd Edition.

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

[13] A. Pettorossi. Transformation of programs and use of tupling strategy. *Proc. Informatica '77, Bled, Yugoslavia*, pp. 1–6, 1977.

[14] A. Pettorossi, M. Proietti, S. Renault. Enhancing partial deduction via unfold/fold rules. *Proc. LoPSTr '96, Stockholm, Sweden.* Springer Verlag, 1996. (To appear).

[15] M. Proietti, A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theo. Comp. Sc.*, 142(1):89–124, 1995.

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

[17] D.A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. *Proc. PEPM '91, New Haven, CT, USA*, SIGPLAN Notices, 26, 9, pp. 62–71. ACM Press, 1991.

[18] H. Tamaki, T. Sato. Unfold/fold transformation of logic programs. *Proc. ICLP '84, Uppsala, Sweden*, pp. 127–138. 1984.

[19] V.F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.