Program Derivation via List Introduction

Alberto Pettorossi * *and Maurizio Proietti* **

**Department of Informatics, University of Roma Tor Vergata,*
Via della Ricerca Scientifica, 00133 Roma, Italy.
email: adp@iasi.rm.cnr.it
***IASI-CNR, Viale Manzoni 30, 00185 Roma, Italy.*
email: proietti@iasi.rm.cnr.it

Abstract

We present a new program transformation strategy based on the introduction of lists. This strategy overcomes some of the limitations of the tupling strategy, and in particular, it allows us to derive linear recursive programs when the tupling strategy is not successful. The linear recursive programs one may derive using this new strategy, have in most cases very good time and space performance. We present our list introduction strategy in the case of definite logic programs, but it can also be applied in the case of functional programs thereby improving some transformation techniques proposed by Cohen (1983) and by Chin and Hagiya (1995).

Keywords

Program transformation, logic programming, unfolding/folding transformation rules, tupling strategy, generalization strategy, list introduction strategy.

1 INTRODUCTION

There are various methodologies for developing programs from specifications, and in particular, for program modules of small size, one can follow the *transformation methodology*, which has been first advocated by Burstall and Darlington (1977) in the case of functional programs. This methodology is very useful in practice, because among other advantages, it can be applied to the derivation of programs written in different programming languages and it can also simplify the proofs of program correctness. In particular, in the so called 'rules + strategies' approach, correctness proofs are avoided because the rules are guaranteed to preserve program semantics.

The transformation methodology has been first applied to logic programs

© IFIP 1997. Published by Chapman & Hall

by Tamaki and Sato (1984), and it has been developed for programs with and without negation in the bodies of the clauses. Different sets of transformation rules have been suggested for the different classes of logic programs under consideration and the different semantics to be preserved. The interested reader may refer to Pettorossi and Proietti (1994) for a survey of recent results in the area of logic program transformation.

In this paper we consider definite logic programs, that is, no negated atoms are allowed in the bodies of the clauses, and we use the following transformation rules described in Tamaki and Sato (1984): (i) *definition introduction*, which allows us to define new predicates, often called eureka predicates in the literature, by the introduction of new clauses, (ii) *unfolding*, which consists of an application of SLD-resolution, (iii) *folding*, which substitutes an instance of the body of a clause by the corresponding instance of the head, and (iv) *goal replacement*, which replaces an old goal by a new goal. In particular, let us consider a clause C of a program P and a goal G_1 in the body of C. We can replace G_1 by the new goal G_2 if $M(P) \models \forall \overline{X}.(\exists \overline{Y}.\ G_1 \leftrightarrow \exists \overline{Z}.\ G_2)$, where $M(P)$ denotes the least Herbrand model of P, $\overline{X}$ is the *tuple* of the variables occurring in the clause obtained from C by removing G_1, $\overline{Y}$ is the tuple of the variables occurring in G_1 and not in $\overline{X}$, and $\overline{Z}$ is the tuple of the variables occurring in G_2 and not in $\overline{X}$.

For the strategies we focus our attention on the tupling and generalization strategies described in Burstall and Darlington (1977), Pettorossi (1977), Darlington (1981), and Pettorossi and Proietti (1994). We first show some of their limitations and we then propose a new strategy, called *list introduction*, to overcome these limitations.

The tupling and generalization strategies

Let us begin by recalling the basic facts about the *tupling strategy*. We are given a clause of the form:

$C.\quad H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$

with $m \geq 1$ and $n \geq 0$. The application of the tupling strategy amounts to the introduction of a new predicate, say t, defined by a clause of the form:

$T.\quad t(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_m$

where the arguments $X_1, \ldots, X_k$ of t are the distinct variables occurring in at least one of the atoms $A_1, \ldots, A_m$ and also occurring elsewhere in C. We then fold clause C w.r.t. the atoms $A_1, \ldots, A_m$ using clause T, and we finally look for the recursive definition of the predicate t by performing some unfolding and goal replacement steps followed by suitable folding steps using clause T itself.

The tupling strategy can be used for transforming C into a set of linear

recursive clauses, that is, a set of clauses such that the body of each of them contains at most one predicate call which depends (possibly via other predicate calls) on the predicate in the corresponding head. In order to achieve this objective via the application of the tupling strategy, the following additional constraint should hold: every folding step using clause T should replace in the body where it is applied, all calls depending on the head predicate by a single call of t. Thus, in particular, the atoms $A_1, \ldots, A_m$ should contain all calls of the head predicate of C.

The application of the tupling strategy may result in an improvement of program performance because in the derived programs we need to evaluate only once the subgoals which are common to the computations evoked by the tupled atoms $A_1, \ldots, A_m$. Moreover, by tupling we can also avoid multiple visits of data structures and the construction of intermediate bindings.

In order to derive a recursive definition of the predicate t introduced by tupling it is often necessary to introduce some extra predicates. This can be done by the *generalization strategy* which we now describe.

Suppose that by performing unfolding and goal replacement steps, from a clause introduced by an application of the tupling strategy, like clause T above, we derive a new clause (possibly, together with other clauses) of the form:

U. $P \leftarrow E_1, \ldots, E_r, F_1, \ldots, F_s$

and from U, by further unfoldings and goal replacements which do not involve $F_1, \ldots, F_s$, we derive:

V. $Q \leftarrow H_1, \ldots, H_r, L_1, \ldots, L_t$

such that, for $i = 1, \ldots, r$, H_i has the same predicate symbol occurring in E_i.

By generalization we may introduce the new predicate g defined by the clause:

G. $g(X_1, \ldots, X_d) \leftarrow J_1, \ldots, J_r$

where (i) the goal $J_1, \ldots, J_r$ is the most specific generalization of both $E_1, \ldots, E_r$ and $H_1, \ldots, H_r$, so that $(J_1, \ldots, J_r)\ \theta_E = (E_1, \ldots, E_r)$ and $(J_1, \ldots, J_r)\ \theta_H = (H_1, \ldots, H_r)$ for some substitutions θ_E and θ_H (as usual, when specifying these substitutions we assume that the clauses U, V, and G have pairwise disjoint sets of variables), and (ii) $X_1, \ldots, X_d$ are (not necessarily all) variables occurring in $J_1, \ldots, J_r$, which allow us to fold both U and V using G (see, for instance, Pettorossi and Proietti (1994)). This last condition amounts to say that a variable, say X, occurring in $J_1, \ldots, J_r$, should be included among the arguments of g iff X/u is a binding in $\theta_E \cup \theta_H$ such that u is *either* (i) a non-variable term, *or* (ii) a variable occurring in one of the atoms $P, F_1, \ldots, F_s, Q, L_1, \ldots, L_t$, *or* (iii) a variable occurring in a term z for some binding Y/z in $\theta_E \cup \theta_H$ with $Y \neq X$.

We then fold clause U using clause G and we look for the recursive definition of predicate g.

We also assume that it is possible to apply the generalization strategy

starting from an initial clause which has been introduced not by tupling, but by a previous application of the generalization strategy itself, like clause G above.

A successful application of the tupling strategy

We illustrate this point by looking at the familiar derivation of a linear-time program for computing the Fibonacci numbers starting from the following exponential-time program.

1. $fib(0, s(0)) \leftarrow$
2. $fib(s(0), s(0)) \leftarrow$
3. $fib(s(s(N)), F) \leftarrow fib(s(N), F1),\ fib(N, F2),\ plus(F1, F2, F)$

with the usual clauses for *plus*. This program requires a number of *plus* operations which is exponential w.r.t. the value of N. We are able to derive a linear-time program by transforming clause 3, which is not linear recursive because of the two calls of *fib* in its body, into a set of linear recursive clauses. We start off by applying the tupling strategy, and we introduce the following clause:

4. $t_fib(N, F1, F2) \leftarrow fib(s(N), F1),\ fib(N, F2)$

in which we tupled together the two atoms $fib(s(N), F1)$ and $fib(N, F2)$ of clause 3.

Now, by folding, we may transform clause 3 into the following clause:

5. $fib(s(s(N)), F) \leftarrow t_fib(N, F1, F2),\ plus(F1, F2, F)$

which is linear recursive. As suggested by the tupling strategy, we now look for a recursive definition of the newly introduced predicate t_fib, as follows. By unfolding clause 4 we get:

6. $t_fib(0, s(0), s(0)) \leftarrow$
7. $t_fib(s(N), F1, F2) \leftarrow fib(s(N), F11),\ fib(N, F12),$
   $plus(F11, F12, F1),\ fib(s(N), F2)$

We have that the predicate *fib* is functional, that is,
$\forall N, X, Y.\ fib(N, X),\ fib(N, Y) \leftrightarrow fib(N, X),\ X = Y$
holds in the least Herbrand model of the given Fibonacci program. Thus, we can apply the goal replacement rule and transform clause 7 into the following one:

8. $t_fib(s(N), F1, F2) \leftarrow fib(s(N), F11),\ fib(N, F12),$
   $plus(F11, F12, F1),\ F11 = F2$

By unfolding clause 8 w.r.t. the atom $F11 = F2$, we get:

9. $t_fib(s(N), F1, F11) \leftarrow fib(s(N), F11),\ fib(N, F12),$
   $plus(F11, F12, F1)$

Now the two calls of *fib* in clause 9 are an instance of the body of clause 4 and, by folding, we get the following linear recursive clause:

10. $t_fib(s(N), F1, F2) \leftarrow t_fib(N, F11, F2),\ plus(F11, F2, F1)$

The final program, made out of clauses 1, 2, 6, and 10, is linear recursive and it requires, as we desired, only a linear number of *plus* operations for evaluating a call of *fib*.

The success of this derivation is due to the fact that by applying the unfolding and goal replacement rules to clause 4, from two calls of the Fibonacci predicate we generate again two calls of that predicate (see clause 9). Thus, by tupling together those two calls as we did in clause 4, we derive, after a folding step, a linear recursive program.

A limitation of the tupling and generalization strategies

In general, for any given initial program it is not the case that by tupling together a *fixed* number of predicate calls, we are able to derive a linear recursive program. Indeed, in the next example, we show that in order to get a linear recursive program we should tuple together an *unbounded* number of predicate calls, that is, a number of calls which depends on the values in the input goal.

Let us consider the following *World Series Odds* problem taken from Aho, Hopcroft, and Ullman (1983), page 312. Two persons, A and B, are playing a sequence of games to see who is the champion, that is, the first to win n games for some given n. We assume that each person has a 0.5 chance of winning any particular game. We use the predicate $p(I, J, K)$ to denote that at the end of the current game, (i) A needs to win I games in the future to be the champion, (ii) B needs to win J games in the future to be the champion, and (iii) K is a rational number between 0 and 1, denoting the probability that A will eventually be the champion. To compute predicate p we may use the following program:

1. $p(0, s(J), 1) \leftarrow$
2. $p(s(I), 0, 0) \leftarrow$
3. $p(s(I), s(J), K) \leftarrow p(I, s(J), K1),\ p(s(I), J, K2),\ ave(K1, K2, K)$

where s denotes the successor function and $ave(K1, K2, K)$ holds iff K is the average of $K1$ and $K2$. To understand clause 3, notice that the probability K of A becoming the champion in the case where A needs to win $s(I)$ games in the future and B needs to win $s(J)$ games in the future, can be computed as follows. Let us consider the case where A wins the next game. In this case $p(I, s(J), K1)$ holds, being $K1$ the probability that A wins the game. In the opposite case where A loses the next game, we have that $p(s(I), J, K2)$ holds. Since the probability that A wins (or loses) the next game is 0.5, then K is the average of $K1$ and $K2$.

In order to derive a linear recursive program we may apply the tupling strategy and introduce the following new predicate:

4. $tp(I, J, K1, K2) \leftarrow p(I, s(J), K1),\ p(s(I), J, K2)$

By unfolding clause 4 w.r.t. the atom $p(I, s(J), K1)$ we get:

5. $tp(0, J, s(0), K2) \leftarrow p(s(0), J, K2)$
6. $tp(s(I), J, K1, K2) \leftarrow p(I, s(J), K3),\ p(s(I), J, K4),$
   $ave(K3, K4, K1),\ p(s(s(I)), J, K2)$

In the body of clause 6 there are three calls of p and we cannot reduce their number to two by exploiting the functionality of p.

Actually, one can prove the following general result: given any sequence S of clauses starting from clause 4 such that any clause is obtained from the previous one by unfolding a p call using clause 3, and applying functionality of p whenever possible, we have that the set of *all* calls of the predicate p in a clause of S is *not* an instance of the set of *all* calls of p in a previous clause of S.

The proof of this result is related to the construction of a dag (see Fig. 1) which has the shape of a Pascal Triangle and it is built as follows: we start from the two initial calls $p(I, s(J), K1)$ and $p(s(I), J, K2)$, and then given any p call, we generate its two son calls by performing an unfolding step using clause 3. Indeed, we have that after any number of unfolding steps starting from clause 4 using clause 3, the set of p calls in the body of every generated clause is (the renaming of) a frontier of that dag. By a *frontier* we mean a set of p calls which is generated from the initial set $\{p(I, s(J), K1),\ p(s(I), J, K2)\}$ after zero or more replacements of a p call by the corresponding two son calls.

By construction the p calls in the dag of Fig. 1 satisfy the following properties: (i) when going from any p call to its left child the first argument is decreased by one and the second argument is unchanged, (ii) when going from any p call to its right child the second argument is decreased by one and the first argument is unchanged, and (iii) the third argument of any p call is the average of the third arguments of its two son calls.

Therefore, by tupling together any fixed number of p calls we will never be able to fold all p atoms and thus, we cannot derive a linear recursive program. Moreover, the generalization strategy does not help because if we replace an atom by a more general one, we do not affect the number of p atoms which are generated during the unfolding process.

The objective of the paper is to show that by suitable generalizations performed according to our *list introduction strategy*, one can indeed obtain a linear recursive program in cases where the tupling strategy does not work such as the ones illustrated by the World Series Odds example we have shown above.

Analogous problems concerning the derivation of linear programs have been considered by other authors for the case of imperative and functional languages in Cohen (1983) and Chin and Hagiya (1995). Cohen's solution is

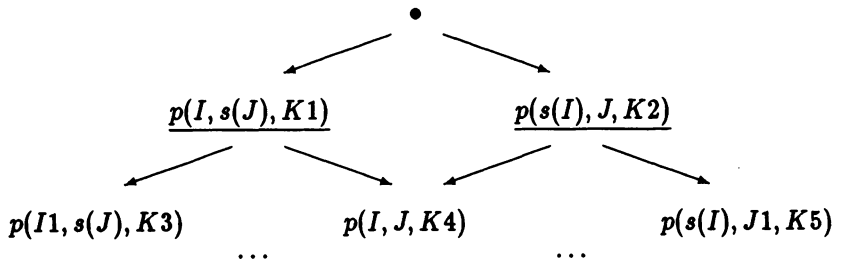

Figure 1 Directed acyclic graph of the p calls generated from the body of clause 4 (see the underlined atoms). This dag has the shape of a Pascal Triangle and the two sons of every p call are generated by unfolding that call using clause 3. We have that $I = s(I1)$ and $J = s(J1)$.

based on some abstract interpretation of the program at hand. By that analysis one determines, given the value of the input, the size of the smallest array needed for storing the results of the different function calls and thus, one avoids all redundant computations. Similarly, in Chin and Hagiya (1995) the proposed solution is based on the use of the memoization technique and a preliminary abstract interpretation of the initial program is necessary for determining the size of the array required.

The approach we describe in this paper is an improvement over those described in Cohen (1983) and in Chin and Hagiya (1995), because it makes use of standard transformational techniques only, and no abstract interpretation of the given program is required. In particular, by using our list introduction strategy together with the tupling and generalization strategies, we are able to derive programs where no repeated computations are performed. As already mentioned, our technique which is based on the introduction of lists of values, works also in the cases when the introduction of arrays of any fixed length is not successful.

Our basic idea can be described as follows. During program transformation we replace a sequence of atoms, say $p(X_1, \ldots), \ldots, p(X_n, \ldots)$, by the single atom $p_list([X_1, \ldots, X_n], \ldots)$ where we have generalized one of the arguments of the predicate p to a list. The definition of the new predicate *p_list* is provided in such a way that redundant information is removed. For instance, all redundant occurrences of the variable Y in the sequence $p(X_1, Y), \ldots, p(X_n, Y)$ are eliminated by replacing that sequence by the atom $p_list([X_1, \ldots, X_n], Y)$ whose recursive definition is as follows:

$p_list([\,], Y) \leftarrow$
$p_list([X|Xs], Y) \leftarrow p(X, Y),\ p_list(Xs, Y)$

We will show that by combining list introduction with tupling and generalization, we are able to derive a final program with the desired performance.

As already mentioned, our technique is presented here with reference to logic programs, but it can easily be applied also in the case of functional programs.

The list introduction strategy can be considered as an extension of the tupling strategy whereby the sequence of atoms $p(X_1, Y), \ldots, p(X_n, Y)$ is replaced by the atom $t(X_1, \ldots, X_n, Y)$ where t is defined by the clause:

$T1.\quad t(X_1, \ldots, X_n, Y) \leftarrow p(X_1, Y), \ldots,\ p(X_n, Y)$

The main advantage of the list introduction strategy over tupling is that by list introduction we are able to replace a sequence of atoms of the form $p(X_1, Y), \ldots,\ p(X_n, Y)$, with $n \geq 0$, by a single atom $p_list([X_1, \ldots, X_n], Y)$, whereas by using the tupling strategy, only sequences of n atoms with predicate p, for some fixed value of n, can be replaced by the corresponding t atom of clause $T1$.

An idea related to the list introduction strategy by which sequences of atoms are encoded into list arguments, was also used in the so-called *compiling control* transformation technique described in Bruynooghe, De Schreye, and Krekels (1989). However, compiling control does not follow the 'rules + strategies' approach, and it needs ad hoc correctness proofs which we do not need here because we rely on the correctness of the transformation rules.

The paper is structured as follows. In the next section we present the derivation of a linear recursive program for the familiar N-queens problem. During this derivation we see the list introduction strategy in action. This strategy is formally defined in Section 3. In that section we also show how list introduction can be combined with tupling and generalization. In Section 4 we show that by our combination of tupling, generalization, and list introduction, it is possible to derive a linear recursive program for solving the World Series Odds problem in quadratic time. Finally, in Section 5 we compare our paper with related work in the area of program transformation and we discuss the issue of mechanizing our proposed list introduction strategy.

2 AN EXAMPLE OF PROGRAM TRANSFORMATION BY LIST INTRODUCTION

Let us consider the familiar N-queens problem: we are required to place N queens on an $N \times N$ board so that no two queens lie on the same horizontal, vertical, or diagonal line. A board configuration with this property is said to be *safe*. Below we will present the initial *N-queens* program, which can be viewed as the formal specification of the given problem. This initial program, similar to the one at page 253 in Sterling and Shapiro (1994), computes the solutions by generating board configurations and checking their safeness.

An $N \times N$ board configuration Qs is represented by a list $[\langle 1, C_1\rangle, \ldots, \langle N, C_N\rangle]$, where each pair $\langle i, C_i\rangle$ for $i \in \{1, \ldots, N\}$ and $C_i \in \{1, \ldots, N\}$ denotes a queen-position in row i and column C_i.

Initial N-queens program

1. $queens(N, Qs) \leftarrow firstnumbers(N, Ns), placequeens(1, Ns, Qs), safeboard(Qs)$
2. $placequeens(R, [\,], [\,]) \leftarrow$
3. $placequeens(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ R+1, placequeens(R1, Ns1, Qs)$
4. $safeboard([\,]) \leftarrow$
5. $safeboard([Q|Qs]) \leftarrow safequeen(Q, Qs),\ safeboard(Qs)$
6. $safequeen(Q, [\,]) \leftarrow$
7. $safequeen(Q1, [Q2|Qs]) \leftarrow notattack(Q1, Q2),\ safequeen(Q1, Qs)$

The definitions of the predicates *firstnumbers*, *select*, and *notattack* are the following ones: $firstnumbers(N, Ns)$ holds iff Ns is the list of the first N positive integers, that is, $Ns = [1, \ldots, N]$; $select(C, Ns, Ns1)$ holds iff the list $Ns1$ is obtained from the list Ns of integers by deleting an occurrence of the integer C; and $notattack(Q1, Q2)$ holds iff the queen-position $Q1$ is not on the same diagonal of the queen-position $Q2$.

Notice that, in each board configuration generated by *placequeens*, only one queen may be placed on a given row or column, and thus, in order to check the safeness property it is only required to verify that no two queens lie on the same diagonal.

Our initial program is a typical application of the *generate-and-test* technique and it is highly inefficient, because many unsafe board configurations are generated. At page 255 of the book by Sterling and Shapiro (1994) the authors propose a more efficient *accumulator* version of the *N-queens* program which uses an accumulator to store partially generated board configurations. This accumulator parameter allows us to check whether or not a queen to be placed on the board, is on the same diagonal of an already placed queen. By doing so, backtracking may occur before an unsafe $N \times N$ board configuration is completely generated.

By applying our proposed transformation technique we will mechanically derive a program which is similar to the accumulator program version. Indeed, the use of list introduction, tupling, and generalization, will allow us to realize the so called *filter promotion* strategy described by Darlington (1978) and Bird (1984), by which the safeness test is 'promoted' into the generation process and the number of generated unsafe board configurations is decreased. A similar effect may also be achieved by the *compiling control* technique described in Bruynooghe, De Schreye, and Krekels (1989), which works by transforming a given initial logic program into a new program whose execution simulates the execution of the initial program under a more sophisticated control strategy.

In this example the filter promotion strategy can be realized by tupling together all calls of the predicates in the body of a clause which act on the same board configuration. This means that we have to apply the tupling

strategy in such a way that the following constraint is satisfied: every folding step performed by using a clause introduced by tupling should produce a clause whose body contains at most one occurrence of a variable denoting a board configuration.

The tupling strategy starts off by introducing the definition of the new predicate r through the following clause:

$R.\quad r(Ns, Qs) \leftarrow placequeens(1, Ns, Qs),\ safeboard(Qs)$

whose body is the conjunction of the atoms occurring in the body of clause 1 and sharing the variable Qs which denotes a board configuration. By folding clause 1 using clause R, we get:

$$queens(N, Qs) \leftarrow firstnumbers(N, Ns),\ t1(Ns, Qs)$$

Now we look for the recursive clauses defining the predicate r. By unfolding clause R w.r.t. the atoms *placequeens* and *safeboard*, we get:

$$r([\,],[\,]) \leftarrow$$
$$r(Ns, [\langle 1, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ 1{+}1,\ placequeens(R1, Ns1, Qs),\ safeboard([\langle 1, C\rangle | Qs])$$

In order to get a clause whose body has only one occurrence of the variable Qs, we would like to fold this last clause we have derived by using clause R. But, unfortunately, this is not possible because '$placequeens(R1, Ns1, Qs)$, $safeboard([\langle 1, C\rangle | Qs])$' is not an instance of the body of that definition. However, to perform that folding step we may apply the generalization strategy, and we may introduce the predicate t by the following clause:

$8.\quad t(N, Ns, Qs) \leftarrow placequeens(N, Ns, Qs),\ safeboard(Qs)$

whose body is the most specific generalization of the goal

$$placequeens(R1, Ns1, Qs),\ safeboard([\langle 1, C\rangle | Qs])$$

we wanted to fold, and the body of clause R.

Now the predicate *queens* can be expressed in terms of the newly introduced predicate t by folding clause 1 using clause 8 whereby deriving the following clause:

$9.\quad queens(N, Qs) \leftarrow firstnumbers(N, Ns),\ t(1, Ns, Qs)$

whose body contains only one occurrence of Qs as desired. However, the body of clause 8 still has more than one occurrence of the variable Qs denoting a board configuration, and therefore no real gain has been achieved by this folding step. Indeed, in order to get a significant efficiency improvement, we need to avoid multiple occurrences of the variable Qs at each level of recursion. This is why, as prescribed by the tupling strategy, the transformation process now proceeds by looking for a recursive definition of the predicate t, such that the body of each clause contains at most one occurrence of any variable denoting a board configuration. In order to get this recursive definition, we will first unfold clause 8 w.r.t. *placequeens*, and then the resulting clause w.r.t. *safeboard*. By doing so we get:

10. $t(R, [\,], [\,]) \leftarrow$
11. $t(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ R+1,$
$placequeens(R1, Ns1, Qs),\ \underline{safequeen(\langle R, C\rangle, Qs)},$
$safeboard(Qs)$

(In this clause and in clause 13 we have underlined some atoms for a reason which we will explain below.) It is now possible to fold clause 11 using clause 8, and by doing so we get the following recursive clause for the predicate t:

12. $t(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ R+1,$
$t(R1, Ns1, Qs),\ safequeen(\langle R, C\rangle, Qs)$

Unfortunately, clause 12 contains two occurrences of the variable Qs denoting a board configuration and the evaluation of the predicate t still has an inefficient generate-and-test behaviour.

To overcome this difficulty, instead of folding clause 11, we continue the derivation process by unfolding clause 11. Indeed, if we unfold clause 11 w.r.t. the *placequeens* atom and then we unfold the derived clause w.r.t. the *safeboard* atom, we derive the following clause:

13. $t(R, Ns, [\langle R, C\rangle, \langle R1, C1\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ R+1,$
$select(C1, Ns1, Ns2),\ R2\ is\ R1+1,$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ placequeens(R2, Ns2, Qs),$
$\underline{safequeen(\langle R, C\rangle, Qs)},\ \underline{safequeen(\langle R1, C1\rangle, Qs)},$
$safeboard(Qs)$

If we now apply the tupling strategy and we fold the goal

$placequeens(R2, Ns2, Qs),\ safeboard(Qs)$

in clause 13 using clause 8, we get two calls of the predicate *safequeen* and a call of t all sharing the variable Qs which denotes a board configuration, and therefore we will not do this folding step.

We may try to continue the program transformation process by applying the generalization strategy. But, unfortunately, there is no pair of goals, say $E_1, \ldots, E_r$ and $H_1, \ldots, H_r$, such that: (i) they occur in the bodies of the clauses 11 and 13, respectively, (ii) they have a common generalization, and (iii) they contain all atoms sharing the variable Qs which occur in the bodies of clauses 11 and 13, respectively.

Although the tupling and the generalization strategies do not work, we may continue our transformation process as we now indicate, and this will provide an example of the list introduction strategy we want to introduce in this paper. We make the key observation that in clause 11 there is *one* atom with predicate *safequeen* and in clause 13, derived from clause 11 by unfolding, there are *two* atoms with predicate *safequeen* (see the underlined atoms). If we continue the unfolding process, we generate a clause with three *safequeen* atoms, and so on.

All these *safequeen* atoms are variants of each other and they share the

second argument *Qs*. As we will show in more detail in the next section, this allows us to apply the list introduction strategy by defining the following new predicate *safequeen_list* which is the so called *list generalization* of the predicate *safequeen*:

14. $safequeen_list([\,], Qs) \leftarrow$
15. $safequeen_list([\langle R, C\rangle | Ps], Qs) \leftarrow safequeen(\langle R, C\rangle, Qs),$
$safequeen_list(Ps, Qs)$

By clause 14 an atom of the form $safequeen_list([\,], Qs)$ can be added to the body of any clause, for any instance of *Qs*. This step can be viewed as an application of the goal replacement rule. Also, by folding using clause 15, any goal which is an instance of '$safequeen(\langle R, C\rangle, Qs)$, $safequeen_list(Ps, Qs)$' can be replaced by the corresponding instance of $safequeen_list([\langle R, C\rangle | Ps], Qs)$. Thus, by using clauses 14 and 15, any conjunction of n atoms, with $n \geq 0$, of the form

$$safequeen(\langle R1, C1\rangle, Qs), \ldots, \; safequeen(\langle Rn, Cn\rangle, Qs)$$

occurring in the body of a clause, can be replaced by the atom

$$safequeen_list([\langle R1, C1\rangle, \ldots, \langle Rn, Cn\rangle], Qs)$$

In particular, clauses 11 and 13 may be transformed into the following clauses 16 and 17, respectively:

16. $t(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1), \; R1 \; is \; R+1,$
$placequeens(R1, Ns1, Qs), \; safequeen_list([\langle R, C\rangle], Qs),$
$safeboard(Qs)$
17. $t(R, Ns, [\langle R, C\rangle, \langle R1, C1\rangle | Qs]) \leftarrow select(C, Ns, Ns1), \; R1 \; is \; R+1,$
$select(C1, Ns1, Ns2), \; R2 \; is \; R1+1,$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle), \; placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R, C\rangle, \langle R1, C1\rangle], Qs),$
$safeboard(Qs)$

We are now able to apply the generalization strategy and we introduce a new predicate *gent* defined by the following clause:

18. $gent(X, Y, Qs, R, C, Ps) \leftarrow placequeens(X, Y, Qs),$
$\underline{safequeen_list([\langle R, C\rangle | Ps], Qs)}, \; safeboard(Qs)$

whose body is the most specific generalization (see the underlined atoms) of the goal

$placequeens(R1, Ns1, Qs), \; \underline{safequeen_list([\langle R, C\rangle], Qs)}, \; safeboard(Qs)$

in the body of clause 16 and the goal

$placequeens(R2, Ns2, Qs), \underline{safequeen_list([\langle R, C\rangle, \langle R1, C1\rangle], Qs)}, safeboard(Qs)$

in the body of clause 17. Then, by folding clause 11 we get:

19. $t(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C, Ns, Ns1),\ R1\ is\ R{+}1,$
$gent(R1, Ns1, Qs, R, C, [\,])$

The program derived so far is made out of clauses 9, 10, 19, together with the clauses which are required for the evaluation of the predicate *gent* defined by clause 18. Thus, we are left with the task of deriving a recursive definition of this predicate.

By unfolding clause 18 w.r.t. *placequeens* we get:

20. $gent(R, [\,], [\,], R, C, Ps) \leftarrow safequeen_list([\langle R, C\rangle | Ps], [\,])$
21. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, Ps) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1{+}1,$
$placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R, C\rangle | Ps], [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

Since $safequeen_list([\langle R, C\rangle | Ps], [\,]) \leftrightarrow true$ in the least Herbrand model of the program at hand (see clauses 14 and 15), by applying goal replacement to clause 20 we get:

20.1 $gent(R, [\,], [\,], R, C, Ps) \leftarrow$

By unfolding clause 21 w.r.t. *safequeen_list* and the resulting clause w.r.t. *safequeen*, we get:

22. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, Ps) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1{+}1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ \underline{safequeen(\langle R, C\rangle, Qs)},$
$safequeen_list(Ps, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

By unfolding clause 22 w.r.t. *safequeen_list* we get:

23. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, [\,]) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1{+}1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ safequeen(\langle R, C\rangle, Qs),$
$safeboard([\langle R1, C1\rangle | Qs])$
24. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, [\langle R3, C3\rangle | Ps1]) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1{+}1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ safequeen(\langle R, C\rangle, Qs),$
$safequeen(\langle R3, C3\rangle, [\langle R1, C1\rangle | Qs]),$
$safequeen_list(Ps1, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

By unfolding clause 24 w.r.t. $safequeen(\langle R3, C3\rangle, [\langle R1, C1\rangle | Qs])$ we have:

25. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, [\langle R3, C3\rangle | Ps1]) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1+1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ \underline{safequeen(\langle R, C\rangle, Qs)},$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle),\ \underline{safequeen(\langle R3, C3\rangle, Qs)},$
$safequeen_list(Ps1, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

Let us now look at the unfolding process from clause 22 to clause 25. Similarly to the unfolding steps which lead from clause 11 to clause 13, we have that in clause 22 there is one atom with predicate *safequeen* and in clause 25 there are two atoms with predicate *safequeen* (see the underlined atoms). All these *safequeen* atoms are variants of each other and they all share the variable Qs denoting a board configuration. Thus, we may apply the list introduction strategy (see Section 3) as follows.

By applying the goal replacement rule we replace the underlined atom of clause 22 by $safequeen_list([\langle R, C\rangle], Qs)$ and the two underlined atoms of clause 25 by the atom $safequeen_list([\langle R, C\rangle, \langle R3, C3\rangle], Qs)$. We get the following clauses:

26. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, Ps) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1+1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),\ safequeen_list([\langle R, C\rangle], Qs),$
$safequeen_list(Ps, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

27. $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, [\langle R3, C3\rangle | Ps1]) \leftarrow$
$select(C1, Ns1, Ns2),\ R2\ is\ R1+1,$
$placequeens(R2, Ns2, Qs),$
$notattack(\langle R, C\rangle, \langle R1, C1\rangle),$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle),$
$safequeen_list([\langle R, C\rangle, \langle R3, C3\rangle], Qs),$
$safequeen_list(Ps1, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

Now we perform a generalization step and we introduce the predicate *ggent* defined by the following clause:

28. $ggent(R2, Ns2, Qs, R, C, Ps1, Ps2, R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R, C\rangle | Ps1], Qs),$
$safequeen_list(Ps2, [\langle R1, C1\rangle | Qs]),$
$safeboard([\langle R1, C1\rangle | Qs])$

The body of this clause is the most specific generalisation of the following two goals E and H occurring in the body of clauses 26 and 27, respectively:

E. $placequeens(R2, Ns2, Qs)$, $safequeen_list([\langle R, C\rangle], Qs)$,
$safequeen_list(Ps, [\langle R1, C1\rangle | Qs])$, $safeboard([\langle R1, C1\rangle | Qs])$

H. $placequeens(R2, Ns2, Qs)$, $safequeen_list([\langle R, C\rangle, \langle R3, C3\rangle], Qs)$,
$safequeen_list(Ps1, [\langle R1, C1\rangle | Qs])$, $safeboard([\langle R1, C1\rangle | Qs])$

Notice that E and H are the sets of atoms in the body of clauses 26 and 27, respectively, depending on the variable Qs.

We can then fold clause 26 using clause 28 and we get:

26.f $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, Ps) \leftarrow$
$select(C1, Ns1, Ns2)$, $R2$ *is* $R1+1$,
$notattack(\langle R, C\rangle, \langle R1, C1\rangle)$,
$ggent(R2, Ns2, Qs, R, C, [\,], Ps, R1, C1)$

The program derived so far is made out of clauses 9, 10, 19, 20.1, 26.f, together with the clauses for the evaluation of the predicate *ggent* defined by clause 28. Thus, we are now left with the task of deriving a recursive definition of this predicate.

This derivation is shown in the Appendix. It is a derivation very similar to the one we have done above for the predicate *gent*. The final program we get is as follows:

N-queens program with accumulators

9. $queens(N, Qs) \leftarrow firstnumbers(N, Ns)$, $t(1, Ns, Qs)$
10. $t(R, [\,], [\,]) \leftarrow$
19. $t(R, Ns, [\langle R, C\rangle | Qs]) \leftarrow select(C,Ns,Ns1)$, $R1$ *is* $R+1$,
$gent(R1,Ns1,Qs,R,C,[\,])$

20.1 $gent(R, [\,], [\,], R, C, Ps) \leftarrow$

26.f $gent(R1, Ns1, [\langle R1, C1\rangle | Qs], R, C, Ps) \leftarrow$
$select(C1, Ns1, Ns2)$, $R2$ *is* $R1+1$,
$notattack(\langle R, C\rangle, \langle R1, C1\rangle)$,
$ggent(R2, Ns2, Qs, R, C, [\,], Ps, R1, C1)$

29.3 $ggent(R2, Ns2, Qs, R, C, Ps1, [\,], R1, C1) \leftarrow$
$gent(R2, Ns2, Qs, R1, C1, [\langle R, C\rangle | Ps1])$

30.3 $ggent(R2, Ns2, Qs, R, C, Ps1, [\langle R3, C3\rangle | Ps2], R1, C1) \leftarrow$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle)$,
$ggent(R2, Ns2, Qs, R3, C3, [\langle R, C\rangle | Ps1], Ps2, R1, C1)$

This program performs much less backtracking than the initial program and its operational behaviour is similar to the *accumulator program* given in Sterling and Shapiro (1994) at page 255. By clause 19 for predicate t, the first queen position $\langle R, C\rangle$ is selected and predicate *gent* is called with its last argument bound to the empty list. This last argument of *gent* stores the board

configuration generated so far, except for the first queen position (whose row and column correspond to the fourth and fifth argument of *gent*, respectively). When a new queen is placed at position $\langle R1, C1\rangle$ and it is not attacked by a queen in position $\langle R, C\rangle$, the predicate *ggent* checks whether or not this queen is attacked by any previously placed queen whose position is in *Ps* (see clause 26.f). In case it is not attacked, the configuration is updated (see clause 29.3), otherwise, by backtracking, a different queen-position is considered. If no position for the new queen is safe, then by backtracking, the position of a previously placed queen, if any, is modified.

As a final remark, notice that we can simplify the final program we have derived by unfolding clause 9 w.r.t. the atom $t(1, Ns, Qs)$. We leave this task to the reader.

3 THE LIST INTRODUCTION STRATEGY

In this section we formally present our list introduction strategy which is used to replace a list of goals by a single atom having a list of values as one of its arguments. We have already mentioned that the list introduction strategy extends the tupling strategy so that, for instance, we may replace a list of atoms of the form:

(1) $p(X_1, Y), \ldots, p(X_n, Y)$

by the single atom $p_list([X_1, \ldots, X_n], Y)$ where the occurrences of Y in the p atoms have been replaced by a single occurrence of Y in the second argument of *p_list*, thereby avoiding the redundant occurrences of the variable Y. We already saw that the predicate *p_list* can be defined as follows:

$$p_list([\,], Y) \leftarrow$$
$$p_list([X|Xs], Y) \leftarrow p(X, Y),\ p_list(Xs, Y)$$

In this case the use of the *p_list* predicate corresponds to the use of the higher-order function *map* in a functional programming language. Indeed, we have that:

$$map\ (\lambda x.p(x, Y))\ [X_1, \ldots, X_n] = [p(X_1, Y), \ldots, p(X_n, Y)]$$

However, it is not always the case that redundant arguments in a list of atoms occur at the same argument position. For instance, let us consider the following list of atoms:

(2) $q(T_0, X_1, T_1), q(T_1, X_2, T_2), q(T_2, X_3, T_3), \ldots, q(T_{n-1}, X_n, T_n)$

where every atom has its third argument equal to the first argument of the atom to its right. In this case we can avoid the intermediate variables $T_1, \ldots, T_{n-1}$ by replacing the given list (2) of atoms by the atom $q_list(T_0, [X_1, \ldots, X_n], T_n)$, defined as follows:

$$q_list(T, [\,], T) \leftarrow$$
$$q_list(T_0, [X|Xs], T_n) \leftarrow q(T_0, X, T_1),\ q_list(T_1, Xs, T_n)$$

The use of the predicate *q_list* is similar to the use of the higher-order function *foldl* when q is a predicate such that $q(T, X, T')$ holds iff $T \oplus X = T'$ for some function $\oplus$. Indeed, we have that: $q_list(T_0, [X_1, \ldots, X_n], T_n)$ holds iff $foldl \ \oplus \ T_0 \ [X_1, \ldots, X_n] = T_n$, where as usual, $foldl \ \oplus \ T \ [\,] = T$ and $foldl \ \oplus \ T_0 \ [X_1, \ldots, X_n] = (\ldots(T_0 \oplus X_1) \ldots \oplus X_n)$.

Now for any given predicate p we provide a general definition of *p_list* which works in particular for lists of atoms of kind (1) and (2) shown above. In this general definition we also consider the case of a list of possibly *non-atomic* goals such that each goal may contain internal variables, that is, variables occurring in that goal only, and not elsewhere in the clause where that goal occurs. In general, a variable X occurring in a goal G in the body of a clause C is said to be an *internal variable of G in C* iff each occurrence of X in C is also an occurrence of X in G.

Recall that by $\overline{X}$ we denote a tuple of variables $\langle X_1, \ldots, X_n \rangle$ for some $n \geq 0$. The empty tuple is denoted by ε. When no confusion arises, we will feel free to identify the variable Z with the singleton tuple $\langle Z \rangle$. Two tuples $\overline{X}$ and $\overline{Y}$ of variables are *disjoint* iff no variable occurs in both of them. In order to denote goals and to refer to the variables occurring in them, we use greek letters with tuple arguments, so that, for instance, $\pi(\overline{V}_1, \ldots, \overline{V}_n)$ denotes a goal such that (i) each of its variables occurs in at least one of the tuples $\overline{V}_1, \ldots, \overline{V}_n$, and (ii) for $i = 1, \ldots, n$, every variable in $\overline{V}_i$ occurs in the goal.

Definition 1. Let $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T}', \overline{Y})$ be a goal where $\overline{I}, \overline{T}, \overline{X}, \overline{T}'$, and $\overline{Y}$ are pairwise disjoint tuples of variables. A goal R occurring in the body of a clause C is said to be a *goal list of length n based on* $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T}', \overline{Y})$ iff it is of the form:

$$\pi(\overline{I}_1, \overline{T}_0, \overline{X}_1, \overline{T}_1, \overline{Y}), \ \pi(\overline{I}_2, \overline{T}_1, \overline{X}_2, \overline{T}_2, \overline{Y}), \ \ldots, \ \pi(\overline{I}_n, \overline{T}_{n-1}, \overline{X}_n, \overline{T}_n, \overline{Y})$$

where: (i) $n \geq 0$, (ii) $\overline{I}_1, \ldots, \overline{I}_n, \overline{T}_0, \ldots, \overline{T}_n, \overline{X}_1, \ldots, \overline{X}_n$, and $\overline{Y}$ are pairwise disjoint tuples of variables, (iii) for $i = 1, \ldots, n$, $\overline{I}_i$ is a tuple of internal variables of $\pi(\overline{I}_i, \overline{T}_{i-1}, \overline{X}_i, \overline{T}_i, \overline{Y})$ in C, and (iv) $\overline{T}_1, \ldots, \overline{T}_{n-1}$ are tuples of internal variables of R in C.

In this definition of a goal list R the order of the arguments is significant. For example, if we interchange the first and fifth arguments of the n variants of the goal $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T}', \overline{Y})$ in R (that is, for $i = 1, \ldots, n$, we interchange $\overline{I}_i$ with $\overline{Y}$), then we obtain a goal R' which is not a goal list. Indeed, in R' the first argument of every π goal is the tuple $\overline{Y}$, while the definition of a goal list requires two different π goals to have two disjoint tuples of variables as first argument.

Notice also that in the definition of a goal list we use five tuples of variables. This is not a restriction, because tuples may consist of any number of variables (even zero).

To see an example of a goal list, let us consider the clause:

$$D.\quad h(U_0,V_0,U_3,V_3,X_1,Y) \leftarrow r(U_0,U_1),\ s(V_1,Y,V_0),\ t(I_1,X_1),\ r(U_1,U_2),\ s(V_2,Y,V_1),\ t(I_2,X_2),\ r(U_2,U_3),\ s(V_3,Y,V_2),\ t(I_3,X_3)$$

The body of D is a goal list of length 3 based on the goal $\delta(I,\langle U,V\rangle,X,\langle U',V'\rangle,Y)$ which is $r(U,U')$, $s(V',Y,V)$, $t(I,X)$. (In the arguments of δ we have identified $\langle I\rangle$, $\langle X\rangle$, and $\langle Y\rangle$ with I, X, and Y respectively.)

In the following definition we present the concept of list generalization. Given a goal $\pi(\overline{I},\overline{T},\overline{X},\overline{T}',\overline{Y})$, by list generalization we introduce a new predicate *p_list* whose second argument is a list.

Definition 2. The *list generalization* of the goal $\pi(\overline{I},\overline{T},\overline{X},\overline{T}',\overline{Y})$ is the predicate *p_list* defined by the following two clauses:

$$C1.\quad p_list(\overline{T},[\,],\overline{T},\overline{Y}) \leftarrow$$
$$C2.\quad p_list(\overline{T}_0,[\overline{X}|Xs],\overline{T}_n,\overline{Y}) \leftarrow \pi(\overline{I},\overline{T}_0,\overline{X},\overline{T}_1,\overline{Y}),\ p_list(\overline{T}_1,Xs,\overline{T}_n,\overline{Y})$$

For instance, the list generalization of the goal $\delta(I,\langle U,V\rangle,X,\langle U',V'\rangle,Y)$ considered above, is the predicate *d_list* defined by the clauses:

$$D1.\quad d_list(\langle U,V\rangle,[\,],\langle U,V\rangle,Y) \leftarrow$$
$$D2.\quad d_list(\langle U_0,V_0\rangle,[X|Xs],\langle U_n,V_n\rangle,Y) \leftarrow r(U_0,U_1),\ s(V_1,Y,V_0),\ t(I_1,X_1),\ d_list(\langle U_1,V_1\rangle,Xs,\langle U_n,V_n\rangle,Y)$$

The clauses for the predicate *p_list* introduced by list generalization can often be simplified by replacing each tuple of k arguments by k distinct arguments and by dropping arguments which are empty tuples.

We will now present some properties of the predicate *p_list* which are used in the derivations presented in this paper.

Property P1. Let P be a program where the predicate *p_list* is defined by clauses $C1$ and $C2$ above. For any goal list R of length n, with $n \geq 0$, based on $\pi(\overline{I},\overline{T},\overline{X},\overline{T}',\overline{Y})$, of the form

$$\pi(\overline{I}_1,\overline{T}_0,\overline{X}_1,\overline{T}_1,\overline{Y}),\ \pi(\overline{I}_2,\overline{T}_1,\overline{X}_2,\overline{T}_2,\overline{Y}),\ \ldots,\ \pi(\overline{I}_n,\overline{T}_{n-1},\overline{X}_n,\overline{T}_n,\overline{Y})$$

we have that:

$$M(P) \models \forall \overline{T}_0,\overline{X}_1,\ldots,\overline{X}_n,\overline{T}_n,\overline{Y}.\ ((\exists \overline{I}_1,\ldots,\overline{I}_n,\overline{T}_1,\ldots,\overline{T}_{n-1}.\ R) \leftrightarrow p_list(\overline{T}_0,[\overline{X}_1,\ldots,\overline{X}_n],\overline{T}_n,\overline{Y}))$$ □

This Property P1 allows us to perform a crucial step when applying the list introduction strategy in the sense that, as specified in the Introduction, we may replace in a clause C the goal list R by the atom $p_list(\overline{T}_0,[\overline{X}_1,\ldots,\overline{X}_n],\overline{T}_n,\overline{Y})$.

Property P2. Let P be a program where the predicate *p_list* is defined by clauses $C1$ and $C2$ above. We have that:

$$M(P) \models \forall \overline{T}_0, L_1, L_2, \overline{T}_n, \overline{Y}.$$
$$((\exists \overline{T}.\ (p_list(\overline{T}_0, L_1, \overline{T}, \overline{Y}),\ p_list(\overline{T}, L_2, \overline{T}_n, \overline{Y}))) \leftrightarrow$$
$$\exists L.\ (append(L_1, L_2, L),\ p_list(\overline{T}_0, L, \overline{T}_n, \overline{Y})))$$

where *append* is the predicate for list concatenation. □

By this equivalence, we may replace in a clause C a goal of the form '$p_list(\overline{T}_0, L_1, \overline{T}, \overline{Y}),\ p_list(\overline{T}, L_2, \overline{T}_n, \overline{Y})$' by the goal '$append(L_1, L_2, L),\ p_list(\overline{T}_0, L, \overline{T}_n, \overline{Y})$'.

By Property P2, the predicate *p_list* defines a *homomorphism* w.r.t. the list structure in its second argument, in the sense of Bird and Meertens (1987).

Let us now describe the list introduction strategy. This strategy combined with the tupling strategy is able to deal with the case where an unbounded number of calls should be tupled together to derive efficient programs. The application of the list introduction strategy essentially consists in: (i) matching a given clause, say U, with a previously derived clause, say V, as in the case of the application of the generalization strategy, (ii) performing some goal replacement steps using Property P1, thereby deriving the new clauses *U_list* and *V_list*, and finally (iii) introducing a new predicate defined by a clause whose body is the most specific generalization of goals occurring in the bodies of *U_list* and *V_list*.

We first introduce the following concept.

Definition 3. Let us consider two goals R_1 and R_2 which are goal lists of length n_1 and n_2, respectively, based on the same goal. We say that the goal R_2 is an *extension* of the goal R_1 iff $n_2 > n_1$.

The List Introduction Strategy

Let us consider the clause C defining a new predicate introduced by applying the tupling or generalization strategies as we have described them in the Introduction. Suppose that by performing unfolding and goal replacement steps, from clause C we derive a clause of the form:

$U.\quad P \leftarrow E_1, \ldots, E_r,\ F,\ H_1, \ldots, H_t$

and from U, by further applications of the unfolding and goal replacement rules not involving $H_1, \ldots, H_t$, we derive:

$V.\quad Q \leftarrow I_1, \ldots, I_r,\ L,\ M_1, \ldots, M_d$

where: (i) for $j = 1, \ldots, r$, E_j and I_j are atoms with the same predicate symbol, (ii) F and L are goal lists based on the same goal, say $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T}', \overline{Y})$, and (iii) L is an extension of F.

Step 1. By list generalization of $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T}', \overline{Y})$ we introduce the new predicate *p_list* defined by clauses $C1$ and $C2$ as in Definition 2.

Step 2. By applying Property P1 we replace F and L by the atoms $p_list(\ldots, \ell_F, \ldots)$ and $p_list(\ldots, \ell_L, \ldots)$, respectively. The values of the list arguments ℓ_F and ℓ_L occurring in the second argument positions in those atoms, are the ones which make Property P1 true. Thus, we derive the following two clauses:

U_list. $P \leftarrow E_1, \ldots, E_r,\ p_list(\ldots, \ell_F, \ldots),\ H_1, \ldots, H_t$

V_list. $Q \leftarrow I_1, \ldots, I_r,\ p_list(\ldots, \ell_L, \ldots),\ M_1, \ldots, M_d$

We have that the length of ℓ_L is strictly greater than the length of ℓ_F, because L is an extension of F.

Step 3. By the generalization strategy, we introduce the new predicate:

$G.\quad genp_list(V_1, \ldots, V_k) \leftarrow N_1, \ldots, N_r,\ p_list(\ldots, \ell, \ldots)$

whose body is the most specific generalization of the goal

$E_1, \ldots, E_r,\ p_list(\ldots, \ell_F, \ldots)$

occurring in the body of *U_list* and the goal

$I_1, \ldots, I_r,\ p_list(\ldots, \ell_L, \ldots)$

occurring in the body of *V_list*. The variables $V_1, \ldots, V_k$ are chosen as already indicated in the generalization strategy described in the Introduction.

Step 4. We fold clause *U_list* using clause G.

Step 5. Finally, we look for the recursive definition of the newly introduced predicate *genp_list*. For deriving this recursive definition we may perform unfolding, folding, and goal replacement steps, including goal replacements based on Properties P1 and P2. □

The reader may verify that the derivation of the N-queens program in Section 2 has been performed according to the combination of the tupling, generalization, and list introduction strategies presented in this section. In particular, clauses 11, 13, 16, 17, and 18 of that derivation correspond, respectively, to the clauses U, V, *U_list*, *V_list*, and G we have now mentioned when describing the five steps of the list introduction strategy.

4 THE WORLD SERIES ODDS EXAMPLE

In this section we return to the World Series Odds example considered in Section 1 and we show that we can derive a quadratic time program from the exponential time specification by using our list introduction strategy together with the tupling and generalization strategies.

For the reader's convenience, we recall here the clauses of the initial program.

1. $p(0, s(J), 1) \leftarrow$
2. $p(s(I), 0, 0) \leftarrow$
3. $p(s(I), s(J), K) \leftarrow p(I, s(J), K1),\ p(s(I), J, K2),\ ave(K1, K2, K)$

We also recall that by applying the tupling strategy we have introduced the following predicate:

4. $tp(I, J, K1, K2) \leftarrow p(I, s(J), K1),\ p(s(I), J, K2)$

By unfolding and goal replacement steps using the functionality of p, from clause 4 we derive:

7. $tp(s(I), s(J), K1, K2) \leftarrow p(I, s(s(J)), K3),\ p(s(I), s(J), K4),$
$p(s(s(I)), J, K5),\ ave(K3, K4, K1),\ ave(K4, K5, K2)$

In the Introduction we have already stated that the application of the tupling strategy cannot be successful in our case, and thus, we may try to apply the list introduction strategy. The application of this new strategy is also suggested by the fact that the number of p atoms is increasing when performing successive unfolding steps.

Notice, however, that the list of p atoms in clause 7 is *not* an extension of the list of p atoms in clause 4 because, in particular, those atoms are not variants of each other. Thus, in order to apply the list introduction strategy we need to derive from clauses 4 and 7 two other clauses which satisfy the conditions for the clauses U and V (see the list introduction strategy in Section 3).

The derivation of these new clauses can be done by using the so called *generalization + equality introduction* rule (see Proietti and Pettorossi (1995)) which is an instance of the goal replacement rule. It consists in replacing a goal, say $A_1, \ldots, A_n$, by the goal $Gen_1, \ldots, Gen_n,\ X_1 = t_1, \ldots, X_r = t_r$, such that $(Gen_1, \ldots, Gen_n)\theta = A_1, \ldots, A_n$, where θ is the substitution $\{X_1/t_1, \ldots, X_r/t_r\}$. Notice that by this rule we promote a term to a variable, and thus, it is always possible to generate a list of p atoms which are variants of each other.

By using the generalization+equality introduction rule we transform clauses 4 and 7 into the following two clauses which satisfy the necessary conditions for applying the list introduction strategy:

8. $tp(I, J, K1, K2) \leftarrow J1 = s(J),\ p(I, J1, K1),\ I1 = s(I),$
$p(I1, J, K2)$

9. $tp(s(I), s(J), K1, K2) \leftarrow J2 = s(J1),\ p(I, J2, K3),\ I1 = s(I),$
$J1 = s(J),\ p(I1, J1, K4),\ I2 = s(I1),$
$p(I2, J, K5),$
$ave(K3, K4, K1),\ ave(K4, K5, K2)$

Indeed, let us consider the goal:

F: $J1 = s(J),\ p(I, J1, K1),\ I1 = s(I)$

in the body of clause 8 and the following goals in the body of clause 9:

L_1: $J2 = s(J1),\ p(I, J2, K3),\ I1 = s(I)$, and

L_2: $J1 = s(J),\ p(I1, J1, K4),\ I2 = s(I1)$

Let $\mu(\varepsilon, \langle J1, I\rangle, K1, \langle J, I1\rangle, \varepsilon)$ be the goal

$J1 = s(J),\ p(I, J1, K1),\ I1 = s(I)$.

The goal F and the goals L_1 and L_2 are all variants of the goal $\mu(\varepsilon, \langle J1, I\rangle, K1, \langle J, I1\rangle, \varepsilon)$. Actually, F and (L_1, L_2) are goal lists based on the same goal $\mu(\varepsilon, \langle J1, I\rangle, K1, \langle J, I1\rangle, \varepsilon)$. Moreover, (L_1, L_2) is an extension of F. Thus, we may apply the list introduction strategy. Here are the corresponding five steps.

Step 1. By list generalization of $\mu(\varepsilon, \langle J1, I\rangle, K1, \langle J, I1\rangle, \varepsilon)$ we define the following predicate:

10. $m_list(J2, I, [\,], J2, I) \leftarrow$
11. $m_list(J1, I, [K1|Ks], Jn, In) \leftarrow J1 = s(J),\ p(I, J1, K1),\ I1 = s(I),$
$m_list(J, I1, Ks, Jn, In)$

where we have replaced tuples of variables by distinct arguments and we have dropped arguments with empty tuples.

Step 2. By Property P1 clauses 8 and 9 can be transformed into the following two clauses, respectively (they correspond to the clauses *U_list* and *V_list* of the list introduction strategy in Section 3):

12. $tp(I, J, K1, K2) \leftarrow m_list(J1, I, [K1], J, I1),\ p(I1, J, K2)$
13. $tp(s(I), s(J), K1, K2) \leftarrow m_list(J2, I, [K3, K4], J, I2),$
$p(I2, J, K5),$
$ave(K3, K4, K1),\ ave(K4, K5, K2)$

Step 3. By the generalization strategy we introduce the following new predicate:

14. $genm_list(J1, I, K1, Ks, J, K2) \leftarrow m_list(J1, I, [K1|Ks], J, I1),$
$p(I1, J, K2)$

whose body is the most specific generalization of the goal:

$m_list(J1, I, [K1], J, I1),\ p(I1, J, K2)$ occurring in the body of clause 12, and the goal

$m_list(J2, I, [K3, K4], J, I2),\ p(I2, J, K5)$ occurring in the body of clause 13.

Step 4. We fold clause 12 by using the newly introduced predicate *genm_list* and we get:

15. $tp(I, J, K1, K2) \leftarrow genm_list(J1, I, K1, [\,], J, K2)$

Step 5. We are now left with the task of finding a recursive definition of *genm_list*. This task may be accomplished by applying for a few more times the list introduction and generalization strategies. For lack of space we will not present here the entire derivation. The final program one may derive is as follows:

1. $p(0, s(J), 1) \leftarrow$
2. $p(s(I), 0, 0) \leftarrow$
3. $p(s(I), s(J), K) \leftarrow tp(I, J, K1, K2),\ ave(K1, K2, K)$

15. $tp(I, J, K1, K2) \leftarrow genm_list(J1, I, K1, [\,], J, K2)$
16. $genm_list(s(A), 0, 1, B, C, D) \leftarrow new1(A, C, s(0), B, D)$
17. $genm_list(s(A), s(B), C, D, E, F) \leftarrow$
$new2(A, B, G, [\,], H, E, D, F),\ ave(G, H, C)$
18. $new1(A, A, B, [\,], C) \leftarrow p(B, A, C)$
19. $new1(s(A), B, 0, [1|C], D) \leftarrow new1(A, B, s(0), C, D)$
20. $new1(s(A), B, s(C), [D|E], F) \leftarrow$
$new2(A, C, G, [\,], H, B, E, F),\ ave(G, H, D)$
21. $new2(0, A, B, C, D, 0, [\,], 0) \leftarrow new1(E, 0, A, [B|C], D)$
22. $new2(s(A), B, C, D, E, s(A), [\,], F) \leftarrow append(D, [E], Z),$
$new1(G, A, B, [C|Z], J),\ ave(E, J, F)$
23. $new2(s(A), B, C, D, E, F, [G|H], I) \leftarrow append(D, [E], Z),$
$new2(A, B, C, Z, M, F, H, I),\ ave(E, M, G)$

This derived program is linear recursive and it can be shown that $O(m \times n)$ calls of *ave* are required for evaluating a goal of the form $p(s^m(0), s^n(0), K)$. Our final program could be simplified by unfolding the call of *tp* in clause 3, and also unfolding the calls of *genm_list* in the derived clause. We may then discard the clauses which define *tp* and *genm_list* (that is, clauses 15, 16, and 17) because they will never be used for the evaluation of a call of *p*.

Notice that Property P2 is indeed used in our World Series Odds program for deriving the clauses which define the predicate *new2*, as it is shown by the occurrences of the *append* predicate in those clauses. The use of the *append* predicate could easily be removed in favour of cheaper *cons* operations by applying standard transformation techniques.

To conclude this section we would like to comment on the fact that some preliminary generalization + equality introduction steps were necessary to transform clauses 4 and 7 before applying the list introduction strategy. The problem of when and how to perform the suitable generalization + equality introduction steps is not trivial and we are currently studying techniques to address this problem in a systematic way.

5 RELATED WORK AND FINAL DISCUSSION

The tupling and generalization strategies are well established techniques for program derivation using the 'rules + strategies' approach as indicated in Burstall and Darlington (1977). In the case of logic programming these strategies work by combining together several predicate calls so that their interactions can be exploited, and their collective evaluation is more efficient than the sequence of evaluations of the single calls in isolation. A limitation of the tupling and generalization strategies is that the number of calls which can be combined together is fixed independently of the input. Basically, the tupling

and generalization strategies correspond to the introduction of *arrays* of fixed length.

The extension of these strategies we propose in this paper, allows us to combine a number of calls which depends on the input, by introducing *lists* of values which encode sequences of predicate calls.

A related idea of encoding sequences of goals by lists of values was also used in the compiling control transformation technique proposed by Bruynooghe, De Schreye, and Krekels (1989). Compiling control works by first generating a symbolic representation of the computation of a given class of calls and then synthesizing a new program from that symbolic computation. Thus, compiling control does not follow the 'rules + strategies' approach, and as already mentioned, it needs ad hoc correctness proofs which we do not need here because we rely on the correctness of the rules. Moreover, our technique for list introduction allows us to avoid redundant information when a sequence of goals is encoded as a list, thereby avoiding useless intermediate data structures. In this sense, our list introduction strategy is also an extension of the strategy for avoiding unnecessary variables presented in Proietti and Pettorossi (1995).

Our transformational approach also contrasts with other approaches considered in the case of functional languages for solving similar problems such as those described in Cohen (1983) and in Chin and Hagiya (1995), where static analysis techniques are used to precompute the sizes of the arrays required depending on the size of the input.

We have shown through some examples, that list introduction can be used to derive linear recursive programs. Obviously, by encoding sequences of goals using lists, we may encode any stack of suspended predicate calls and by doing so, any program can be transformed into a program where all clauses have at most one call in their body (and therefore they are linear recursive). Thus, our transformation technique is related to the *continuation passing style transformation* proposed by Wand (1980) for functional programs. It is also related to the transformation into binary form proposed for logic programs by Tarau and Boyer (1990), where the stack of suspended computations is encoded as an argument. However, as already mentioned, our list introduction strategy may avoid some redundant computations and, as shown by our examples, it may allow for exponential speedups.

Various issues naturally arise when looking for an efficient mechanization of our technique based on the tupling, generalization, and list introduction strategies. These issues are common to many methods for logic program derivation by transformation and in particular, during the derivation process we have to choose: (i) the atoms to be tupled together when introducing the new eureka definition at the beginning of the transformation process, (ii) the atoms to unfold during program derivation, (iii) the two clauses to be considered for the application of the generalization strategy or the list introduction strategy, and finally, (iv) the sequences of goals to be replaced by a single atom when applying list introduction.

It is not easy to present general and powerful techniques for addressing all these issues in a completely satisfactory way, and we can only offer partial answers for the time being.

For to Point (i) one suitable suggestion is to tuple in the new definition the atoms which have common arguments or have arguments with shared variables. Indeed, if one gets a recursive definition of that newly introduced predicate by folding, it is often the case that the construction of intermediate data structures is avoided and the generate-and-test behaviour of the derived program is improved, similarly to what one can achieve by using the filter promotion technique (see, for instance our N-queens example in Section 2). Another suggestion is to tuple together all calls which occur in the body of a clause and depend on the head predicate of that clause so to derive a linear recursive clause (see, for instance our World Series Odds example in Section 4). More generally, a suitable form of the initial eureka definition may be decided after analyzing the program at hand (see, for instance, the work by Cohen (1983)).

For Point (ii) and Point (iii) much work has been done in the framework of several related techniques, such as *supercompilation* in Turchin (1986), *deforestation* in Wadler (1990), the automation of the tupling strategy in Chin (1993), *partial evaluation* in Jones, Gomard and Sestoft (1993), as well as the already mentioned techniques for compiling control and avoiding unnecessary variables. Since the list introduction strategy is, in principle, independent of the specific way in which we specify the unfolding and the generalization steps, we may use many results presented in the above mentioned papers.

Finally, for Point (iv) we would like to notice that, for a given clause, there may be several ways, possibly conflicting ones, of applying the list introduction strategy and replacing a sequence of goals by a single atom. It is hard to define general policies to choose among these conflicting options. However, in practice, it is often quite simple to make these choices because there are suitable restrictions both on the classes of programs under consideration and the syntactic forms of the programs to be derived by transformation.

APPENDIX

Derivation of the recursive definition of the predicate *ggent* defined by clause 28.

By unfolding clause 28 w.r.t. $safequeen_list(Ps2, [\langle R1, C1\rangle | Qs])$ we get clauses 29 and 30:

29. $ggent(R2, Ns2, Qs, R, C, Ps1, [\,], R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R, C\rangle | Ps1], Qs),$
$safeboard([\langle R1, C1\rangle | Qs])$

30. $ggent(R2, Ns2, Qs, R, C, Ps1, [\langle R3, C3\rangle|Ps2], R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R, C\rangle|Ps1], Qs),$
$safequeen(\langle R3, C3\rangle, [\langle R1, C1\rangle|Qs]),$
$safequeen_list(Ps2, [\langle R1, C1\rangle|Qs]),$
$safeboard([\langle R1, C1\rangle|Qs])$

By unfolding clause 29 w.r.t. $safeboard([\langle R1, C1\rangle|Qs])$ we get:

29.1 $ggent(R2, Ns2, Qs, R, C, Ps1, [\,], R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$\underline{safequeen_list([\langle R, C\rangle|Ps1], Qs)},$
$\underline{safequeen(\langle R1, C1\rangle, Qs)},\ safeboard(Qs)$

By folding the underlined atoms in clause 29.1 using clause 15, we get:

29.2 $ggent(R2, Ns2, Qs, R, C, Ps1, [\,], R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$safequeen_list([\langle R1, C1\rangle, \langle R, C\rangle|Ps1], Qs),$
$safeboard(Qs)$

By folding the whole body of clause 29.2 using clause 18, we get:

29.3 $ggent(R2, Ns2, Qs, R, C, Ps1, [\,], R1, C1) \leftarrow$
$gent(R2, Ns2, Qs, R1, C1, [\langle R, C\rangle|Ps1])$

By unfolding clause 30 w.r.t. $safequeen(\langle R3, C3\rangle, [\langle R1, C1\rangle|Qs])$:

30.1 $ggent(R2, Ns2, Qs, R, C, Ps1, [\langle R3, C3\rangle|Ps2], R1, C1) \leftarrow$
$placequeens(R2, Ns2, Qs),$
$\underline{safequeen_list([\langle R, C\rangle|Ps1], Qs)},$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle),$
$\underline{safequeen(\langle R3, C3\rangle, Qs)},$
$safequeen_list(Ps2, [\langle R1, C1\rangle|Qs]),$
$safeboard([\langle R1, C1\rangle|Qs])$

By folding the underlined atoms in clause 30.1 using clause 15, we get:

30.2 $ggent(R2, Ns2, Qs, R, C, Ps1, [\langle R3, C3\rangle|Ps2], R1, C1) \leftarrow$
$\underline{placequeens(R2, Ns2, Qs)},$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle),$
$\underline{safequeen_list([\langle R3, C3\rangle, \langle R, C\rangle|Ps1], Qs)},$
$\underline{safequeen_list(Ps2, [\langle R1, C1\rangle|Qs])},$
$\underline{safeboard([\langle R1, C1\rangle|Qs])}$

By folding the underlined atoms in clause 30.2 using clause 28, we get:

30.3 $ggent(R2, Ns2, Qs, R, C, Ps1, [\langle R3, C3\rangle|Ps2], R1, C1) \leftarrow$
$notattack(\langle R3, C3\rangle, \langle R1, C1\rangle),$
$ggent(R2, Ns2, Qs, R3, C3, [\langle R, C\rangle|Ps1], Ps2, R1, C1).$

ACKNOWLEDGMENTS

We would like to thank D. De Schreye for many fruitful conversations and for comments on the N-queens example presented in this paper. S. Renault implemented an interactive program transformation system which helped us for the development of the examples. 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] Aho, A., Hopcroft, J.E. and Ullman, J.D. (1983) *Data Structures and Algorithms.* Addison-Wesley.

[2] Bird, R.S. (1984) The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504.

[3] Bird, R.S. and Meertens, L.G.L.T. (1987) Two Exercises Found in a Book on Algorithmics. In L.G.L.T. Meertens, editor, *TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, Germany*, 451–457. North-Holland.

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

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

[6] Chin, W.-N. (1993) Towards an automated tupling strategy. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, 119–132. ACM Press.

[7] Chin, W.-N. and Hagiya, M. (1995) A transformational method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115.

[8] Cohen, N.H. (1983) Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299.

[9] Darlington, J. (1978) A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30.

[10] Darlington, J. (1981) An experimental program transformation system. *Artificial Intelligence*, 16:1–46.

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

[12] Pettorossi, A. (1977) Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia*, 1–6.

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

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

[15] Sterling, L.S. and Shapiro, E. (1994) *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts. Second Edition.

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

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

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

[19] Wadler, P.L. (1990) Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73, 231–248.

[20] Wand, M. (1980) Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180.

6 BIOGRAPHIES

Alberto Pettorossi received his laurea degree in Electronic Engineering from La Sapienza University, Rome (Italy), in 1971. From that university he also obtained in 1972 the Specialization degree in Control Systems and Computing. From 1975 until 1988 he was a researcher of the Italian National Research Council at IASI in Rome. In 1979 he obtained the Master degree in Computer Science from Syracuse University, NY, USA. In 1984 he obtained the PhD degree in Computer Science from Edinburgh University, UK. Since 1988 he is an associate professor of Computer Science at the University of Rome Tor Vergata, Italy.

Maurizio Proietti received his laurea degree in Mathematics from La Sapienza University, Rome (Italy), in 1984. He joined the National Research Council of Italy (CNR) in 1988 and he currently works as a researcher at the Institute of Analysis of Systems and Informatics (IASI) in Rome. His research interests include program transformation, specialization, synthesis, and verification in the context of functional and logic programming.