ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Pettorossi, M. Proietti

THE LIST INTRODUCTION STRATEGY FOR THE AUTOMATIC DERIVATION OF PROGRAMS

R. 472 Ottobre 1998

Alberto Pettorossi - Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via di Tor Vergata, I-00133 Roma, Italy. Email : `adp@iasi.rm.cnr.it`. URL : `http://www.iasi.rm.cnr.it/~adp`.

Maurizio Proietti - Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email : `proietti@iasi.rm.cnr.it`. URL : `http://www.iasi.rm.cnr.it/~proietti`.

This work has been partially supported by MURST Progetto Cofinanziato 'Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software' (Italy), Progetto Coordinato CNR 'Programmazione Logica' (Italy), Progetto Coordinato CNR 'Verifica, Analisi e Trasformazione dei Programmi Logici' (Italy), Programma Galileo CRUI (Italy), and INTAS Project 'Efficient Symbolic Computing' 93–1702.

Istituto di Analisi dei Sistemi ed Informatica, CNR
viale Manzoni 30
00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

We present a new program transformation strategy based on the introduction of lists. This strategy is an extension of the tupling strategy which is based on the introduction of tuples of fixed length. The list introduction 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 we may derive using this new strategy, have in most cases very good time and space performance because they avoid repeated evaluations of goals. 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 [8] and by Chin and Hagiya [7].

Key words: Automatic program derivation, program transformation, logic programming, transformation rules, transformation strategies

1. Introduction

There are various methodologies for developing programs from specifications, and in particular, for small program modules one can follow the *transformation methodology*, which has been first advocated by Burstall and Darlington [5] 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 in different language paradigms 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 based on the unfolding and folding rules has been first applied to logic programs by Tamaki and Sato [17], 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 [13] for a survey of 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 four transformation rules described in [17, 13]:
(1) the *definition introduction* rule which allows us to define a new predicate by the introduction of one or more (possibly recursive) new clauses,
(2) the *unfolding* rule which consists of an application of SLD-resolution,
(3) the *folding* rule which substitutes an instance of the body of a clause by the corresponding instance of the head, and
(4) the *goal replacement* rule which replaces an old goal by a new goal (in this paper we use the term 'goal' to mean 'conjunction of atoms').

For the application of the folding rule some technical conditions should hold (see the conditions for the *single-folding* rule in [13]). They are equivalent to the requirement that if we fold a given clause C thereby deriving a clause D, then by unfolding clause D we should obtain a variant of clause C itself.

The goal replacement rule is applied as follows. 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 $M(P) \models \forall X_1, \ldots, X_k.\, (\exists Y_1, \ldots, Y_m.\, G_1 \leftrightarrow \exists Z_1, \ldots, Z_n.\, G_2)$, where: (i) $M(P)$ denotes the least Herbrand model of P, (ii) $\{X_1, \ldots, X_k\}$ is the intersection of the set of variables occurring in (G_1, G_2) and the set of variables occurring in the clause obtained from C by removing G_1, (iii) $\{Y_1, \ldots, Y_m\}$ is the set of variables occurring in G_1 and not in $\{X_1, \ldots, X_k\}$, and (iv) $\{Z_1, \ldots, Z_n\}$ is the set of variables occurring in G_2 and not in $\{X_1, \ldots, X_k\}$.

For the strategies we focus our attention on the tupling and generalization strategies described in [5], [12], [10], and [13]. We first recall these strategies, we then show some of their limitations, and we finally 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 (the order of the atoms in the body of a clause is immaterial because we assume the least Herbrand model semantics):

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

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 set $\{X_1,\ldots,X_k\}$ of the arguments of t is the intersection of the set of variables occurring in the goal $(A_1,\ldots,A_m)$ and the set of variables occurring in the goal $(H,B_1,\ldots,B_n)$. We then fold clause C w.r.t. the goal $(A_1,\ldots,A_m)$ 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 (see [13] for details).

The tupling strategy can be used, for instance, for transforming C into a set of *linear* recursive clauses, that is, a set S of clauses such that the body of each clause, say D, in S contains at most one predicate call which depends (possibly via other predicate calls) on the predicate in the head of clause D itself. 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, all calls of the head predicate of C should be contained in the goal $(A_1,\ldots,A_m)$.

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 [14].

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

Suppose that during program derivation we get a clause (possibly, together with other clauses) of the form:

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

and from U, by unfolding and/or goal replacement steps which do *not* involve $F_1,\ldots,F_s$, we derive:

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

such that, for $i=1,\ldots,r$, the atom H_i has the same predicate symbol which occurs in E_i. During the derivation from clause U to clause V the atoms $F_1,\ldots,F_s$ may get instantiated because of unifications and we assume that those instantiated atoms occur in clause V among the L_i's.

Now let us assume, without loss of generality, that clauses U and V have pairwise disjoint sets of variables. The application of the generalization strategy amounts to the introduction of the new predicate g defined by the clause:

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

where $(J_1,\ldots,J_r)$ is a most specific generalization of $(E_1,\ldots,E_r)$ and $(H_1,\ldots,H_r)$ with no variables in common with $(E_1,\ldots,E_r,H_1,\ldots,H_r)$, and the set $\{X_1,\ldots,X_d\}$ of the arguments of g is chosen as we will specify below.

By construction of clause G, there exist two substitutions, say θ_E and θ_H, such that: (i) the domain of each of them is the set of variables in $(J_1,\ldots,J_r)$, (ii) $(J_1,\ldots,J_r)\,\theta_E=(E_1,\ldots,E_r)$, and (iii) $(J_1,\ldots,J_r)\,\theta_H=(H_1,\ldots,H_r)$.

The set $\{X_1,\ldots,X_d\}$ is constructed according to the following Condition (α): a variable, say X, occurring in $(J_1,\ldots,J_r)$, should be included in that set if X/u is a binding in $\theta_E \cup \theta_H$ and u is *either* (1) a non-variable term, *or* (2) a variable occurring in one of the atoms

$P, F_1, \ldots, F_s, Q, L_1, \ldots, L_t$, *or* (3) a variable occurring in a term z for some binding Y/z in $\theta_E \cup \theta_H$ with $Y \neq X$.

After the introduction of clause G, we fold clause U using clause G and we then look for the recursive definition of predicate g. This concludes the presentation of the generalization strategy.

Notice that the above Conditions (i), (ii), (iii), and (α) allow us to fold clause U using clause G. Actually, they allow us to fold also clause V using G. Indeed, after folding clauses U and V, by unfolding the two derived clauses w.r.t. the atom with predicate g, we get again, modulo variable renaming, clause U and V, respectively. Thus, the extra conditions on the applicability of the folding rule we mentioned earlier on in this section, are satisfied.

Notice also that in the above presentation of the generalization strategy we have left unspecified how to choose clause U, clause V, the E_i's atoms in clause U, and the H_i's atoms in clause V. Different choices may lead to different versions of the generalization strategy, whose detailed study is outside the scope of this paper.

In the examples given in the paper, when applying the generalization strategy we do not explicitly indicate the variable renaming steps because as usual, we may identify two variant clauses.

The generalization strategy is used during program derivation because it may facilitate the task of getting, via folding steps, the recursive definitions of the new predicates we introduce. Indeed, if starting from clause G we can perform the same sequence of transformation steps we have performed starting from clause U, then when reaching the clause corresponding to V, it is often the case that we can perform a folding step and derive a recursive definition of the predicate g. This is due to the fact that clause G generalizes both clause U and V.

A successful application of the tupling strategy

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

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

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 (due to 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. $t\_\mathit{fib}(N, F1, F2) \leftarrow \mathit{fib}(s(N), F1),\ \mathit{fib}(N, F2)$

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

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

5. $\mathit{fib}(s(s(N)), F) \leftarrow t\_\mathit{fib}(N, F1, F2),\ \mathit{plus}(F1, F2, F)$

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. $t\_\mathit{fib}(0, s(0), s(0)) \leftarrow$

6.

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 in the sense that
$\forall N, X, Y.\ fib(N, X),\ fib(N, Y) \leftrightarrow fib(N, X),\ X = Y$
holds in the least Herbrand model of the given Fibonacci program, where we assume that the equality predicate '=' is defined by the clause: $X = X \leftarrow$. 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. $t\_fib(s(N), F1, F2) \leftarrow fib(s(N), F2),\ fib(N, F12),\ plus(F2, F12, F1)$

Now the pair of calls of *fib* in clause 9 is an instance (actually, a variant) of the body of clause 4 and, by folding, we get the following linear recursive clause:

10. $t\_fib(s(N), F1, F2) \leftarrow t\_fib(N, F2, F12),\ plus(F2, F12, F1)$

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 any two calls of the Fibonacci predicate such as those in clause 4, 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 as we managed in the Fibonacci example above. Indeed, in the next example, we show that in order to get a linear recursive program we should tuple together a *variable* 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 [1], page 312. Two teams, say A and B, are playing a sequence of games: the first to win n games, for some given n, becomes the champion. We assume that each team has probability $1/2$ of winning any particular game in the sequence. We use the atom $p(I, J, K)$ to denote that A has probability K of becoming the champion when A needs to win I games in the future to become the champion, and B needs to win J games in the future to become the champion. The value of K is a rational number between 0 and 1. To evaluate the atom $p(I, J, K)$ at the end of any game in the sequence, 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 $K = (K1 + K2)/2$. Clause 1 says that A has probability 1 of becoming the champion if it needs to win 0 games in the future and B needs to win more than 0 games in the future. Analogously, clause 2 says that A has probability 0 of becoming the champion if it needs to win more than 0 games in the future and B needs to win 0 games in the future. Clause 3 says that the probability K of A becoming the champion when A needs to win $s(I)$ games in the future and B needs to win $s(J)$ games in the future, can be recursively computed as follows. Let us consider the case where A wins

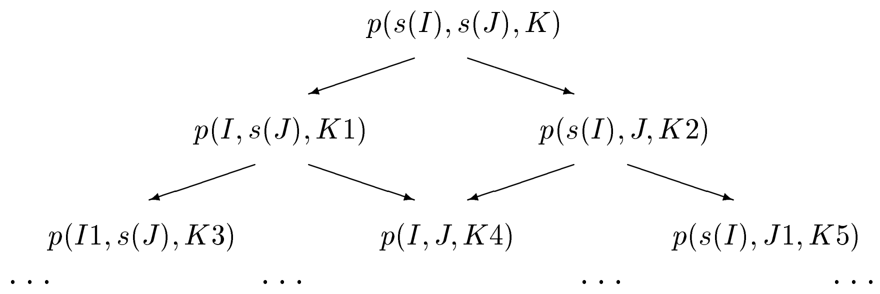

Figure 1: Directed acyclic graph of the p calls generated from the head of clause 3 (see the atom at the root). The two sons of every p call are generated by unfolding that p call using clause 3. We have that $I = s(I1)$ and $J = s(J1)$.

the next game: in this case $p(I, s(J), K1)$ holds for some probability $K1$. In the opposite case where A loses the next game, we have that $p(s(I), J, K2)$ holds for some probability $K2$. Since the probability that A wins (or loses) the next game is $1/2$, we have that K is $(K1 + K2)/2$.

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 in S 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 the sequence 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 based on the construction of a dag (see Fig. 1) which has the shape of a binary tree such that the left son of the right son of any given node is also the right son of the left son of that node. That dag is built as follows: we start from the initial p call which is the atom $p(s(I), s(J), K1)$, and then given any p call, we generate its two son calls by performing an unfolding step using clause 3. 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 given by (the renaming of) a *frontier* of the dag in Fig. 1. By a frontier of that dag we mean a set of p calls which is generated from the singleton set $\{p(s(I), s(J), K1)\}$ after replacing zero or more times 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.

As a consequence of these properties, we have that by tupling together any fixed number of p calls we will never be able to fold all p atoms in any given clause of the sequence S 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 modify 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 (alone or together with the generalization strategy) does not work, for instance, in the World Series Odds example we have shown above.

Analogous problems concerning the derivation of linear programs have been considered by other authors in the case of imperative and functional languages [8, 7]. Cohen's solution is 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 by doing so one may avoid all redundant computations. Similarly, the solution of Chin and Hagiya [7] 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 described in this paper makes use of purely transformational techniques, and no abstract interpretation or memoization 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.

The basic idea of our technique can be described as follows. During program transformation we replace a conjunction 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 the first argument of the predicate p to a list. Moreover, 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 conjunction $p(X_1, Y), \ldots, p(X_n, Y)$ are eliminated by replacing that conjunction by the atom $p\_list([X_1, \ldots, X_n], Y)$ whose recursive definition is as follows:

$L1.\ \ p\_list([\,], Y) \leftarrow$
$L2.\ \ p\_list([X|Xs], Y) \leftarrow p(X, Y),\ p\_list(Xs, Y)$

We will show that by combining list introduction with tupling and generalization, we are able to derive very efficient programs. 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 in the following sense. By using the tupling strategy the goal $p(X_1, Y), \ldots, p(X_n, Y)$ can be replaced by the atom $t(X_1, \ldots, X_n, Y)$ where t is defined by the clause:

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

where the arity of t depends on n, while in the list introduction strategy the same goal $p(X_1, Y)$, $\ldots,\ p(X_n, Y)$ can be replaced by the atom $p\_list([X_1, \ldots, X_n], Y)$ whose definition is given by the above clauses $L1$ and $L2$ and the arity of p_list does *not* depend on n.

An idea related to the list introduction strategy by which conjunctions of atoms are encoded into list arguments, was also used in the so-called *compiling control* transformation technique described in [4]. 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 providing a solution to the World Series Odds problem which requires quadratic time only. Finally, in Section 5 we compare the strategy we propose in this paper with related work in the area of program transformation and we discuss the issue of how to mechanize our 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 [16], computes the solutions by generating board configurations and checking their safeness.

An $n \times n$ board configuration Qs is represented by a list of pairs of the form:

$[\langle R_1, C_1 \rangle, \ldots, \langle R_n, C_n \rangle]$

where for $i = 1, \ldots, n$, the element $\langle R_i, C_i \rangle$ denotes a queen-position in row R_i and column C_i. For $i = 1, \ldots, n$, the values of R_i and C_i belong to the set $\{1, \ldots, n\}$.

Initial n-queens program

1. $queens(Ns, Qs) \leftarrow placequeens(Ns, Qs),\ safeboard(Qs)$
2. $placequeens([\,], [\,]) \leftarrow$
3. $placequeens(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),\ placequeens(Ns1, Qs)$
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 first argument Ns of the predicate *queens* is given in input as the list $[1, \ldots, n]$ and it can be understood as the list of columns which are initially available for positioning the queens. The second argument Qs is computed in output and it is a safe board configuration.

We assume that the predicate $notattack(Q1, Q2)$ holds iff the queen-position $Q1$ is not on the same diagonal of the queen-position $Q2$. The test that the queen-positions are not on the same row or column, can be avoided by assuming the following definition of the *select* predicate: $select(Q, Ns, Ns1)$ holds iff Q is the queen-position $\langle R, C \rangle$ such that row R is the length of Ns, column C is a member of Ns, and $Ns1$ is the list obtained from Ns by deleting the occurrence of C. Indeed, for this choice of the *select* predicate, we have that any board configuration Qs generated by the evaluation of $placequeens(Ns, Qs)$ starting from the initial value $[1, \ldots, n]$ of the list Ns, is made of queen-positions which do not share the same row or column (note that the length of the list Ns decreases by one at each recursive call of *placequeens*). In particular, board configurations with $k \leq n$ queens are of the form: $[\langle n, c_1 \rangle, \langle n-1, c_2 \rangle, \ldots, \langle n-k+1, c_k \rangle]$ where $c_1, c_2, \ldots, c_k$ are distinct members of the list $[1, \ldots, n]$.

Thus, in order to place n queens on an $n \times n$ board so that the resulting configuration is safe, it is enough: (i) to generate a board configuration, say Qs, via the predicate *placequeens*, and then, (ii) to verify that in the configuration Qs no two queens lie on the same diagonal. These tasks are specified by clause 1.

Our initial program is a typical application of the *generate-and-test* technique and it is inefficient, because many unsafe board configurations are generated. At page 255 of [16] a more efficient *accumulator* version of the n-*queens* program is proposed. In this version an accumulator is used to store partially generated board configurations, and this accumulator 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 complete $n \times n$ board configuration is generated, and thus, efficiency is improved.

By applying our proposed transformation technique we will mechanically derive a program which is similar to the accumulator program version of [16]. Indeed, the use of list introduction, tupling, and generalization, will allow us to realize the so called *filter promotion* strategy described in [9] and [2], 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 [4], which works by transforming a given initial 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 applying the tupling strategy with the following constraints: ($\tau 1$) we tuple together *all* calls of the predicates which occur in the body of a clause and act on a board configuration (for board configurations we use the variable names Qs, Ps, and $Ps1$), and ($\tau 2$) the *notattack* atom should occur to the left of the calls which are tupled together.

In a left-to-right mode of evaluation, as in Prolog, these constraints avoid an inefficient generate-and-test behaviour because they force the tests for safeness to be performed also for incomplete board configurations.

The tupling strategy starts off by considering clause 1 and introducing the definition of the new predicate t through the following clause:

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

whose body is the conjunction of all 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 8, we get:

8.f $queens(Ns, Qs) \leftarrow t(Ns, Qs)$

Now we look for the recursive clauses defining the predicate t. By unfolding clause 8 w.r.t. the atoms *placequeens* and *safeboard*, we get:

9. $t([\,],[\,]) \leftarrow$

10. $t(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),\ placequeens(Ns1, Qs),$
$\qquad safequeen(Q, Qs),\ safeboard(Qs)$

Now we cannot fold clause 10 using clause 8 for deriving the recursive clause of the predicate t because the constraint ($\tau 1$) is not satisfied. Indeed, by doing that folding step we would get:

$$t(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),\ t(Ns1, Qs),\ safequeen(Q, Qs)$$

and this clause contains two atoms with the argument Qs which denotes a board configuration.

Thus, instead of folding clause 10 using clause 8, we may apply the tupling strategy. We introduce the new predicate $t1$ by the clause:

11. $t1(Q, Ns, Qs) \leftarrow placequeens(Ns, Qs),\ \underline{safequeen(Q, Qs)},\ safeboard(Qs)$

(In this clause and in clause 13 we have underlined some atoms for a reason which we will explain below.) We then fold clause 10 using clause 11 whereby deriving:

10.f $t(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),\ t1(Q, Ns1, Qs)$

Now we have to look for the recursive clauses defining the predicate $t1$. Starting from clause 11, if we perform unfolding steps w.r.t. the atoms *placequeens*, *safeboard*, and *safequeen*, we get:

12. $t1(Q, [\,], [\,]) \leftarrow$
13. $t1(Q, Ns, [Q1|Qs]) \leftarrow select(Q1, Ns, Ns1),\ notattack(Q, Q1),\ placequeens(Ns1, Qs),$
$\underline{safequeen(Q, Qs)},\ \underline{safequeen(Q1, Qs)},\ safeboard(Qs)$

Again, we cannot fold clause 13 using clause 11 because there are two atoms with the predicate *safequeen* which share the variable Qs, while by using clause 11 we can fold one of them only. Thus, instead of folding, we may apply the tupling strategy again. Unfortunately, no matter how often we continue to apply the tupling strategy, it will always be impossible to fold all atoms with the variable Qs without introducing a new predicate, because it is the case that the number of the *safequeen* atoms increases by one after any sequence of unfolding steps corresponding to those which led us from clause 11 to clause 13. The situation here is analogous to the one we have described in the World Series Odds example and as in that case the tupling strategy cannot be successful.

Moreover, as explained in that example, also the generalization strategy cannot be successful in our derivation.

In order to overcome this difficulty, we may continue our transformation process as indicated below, and this will give an example of the list introduction strategy we want to introduce in this paper. The application of this strategy is based on these observations we already made: (i) in clause 11 there is *one* atom with predicate *safequeen*, (ii) in clause 13 there are *two* atoms with predicate *safequeen* (see the underlined atoms), and (iii) 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 all share the second argument Qs. As we will show in the next section, this allows us to apply the list introduction strategy by defining the following new predicate *safequeen_list* which is said to be the *list generalization* of the atom $safequeen(Q, Qs)$ (see Definition 2 in Section 3):

14. $safequeen\_list([\,], Qs) \leftarrow$
15. $safequeen\_list([Q|Ps], Qs) \leftarrow safequeen(Q, Qs),\ safequeen\_list(Ps, Qs)$

By applying the goal replacement rule (see Property P1 in Section 3), any conjunction of n atoms, with $n \geq 0$, of the form

$$safequeen(Q1, Qs), \ldots,\ safequeen(Qn, Qs)$$

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

$$safequeen\_list([Q1, \ldots, Qn], Qs)$$

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

16. $t1(Q, Ns, Qs) \leftarrow placequeens(Ns, Qs),\ \underline{safequeen\_list([Q], Qs)},\ safeboard(Qs)$

26.f $t(Q, Ns, [Q1|Qs]) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{notattack}(Q, Q1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \underline{\mathit{safequeen\_list}([Q, Q1], Qs)},\ \mathit{safeboard}(Qs)$

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

18. $\mathit{gent}(Ns, Qs, Q, Ps) \leftarrow \mathit{placequeens}(Ns, Qs), \underline{\mathit{safequeen\_list}([Q|Ps], Qs)}, \mathit{safeboard}(Qs)$

whose body is a most specific generalization of the goal

$\mathit{placequeens}(Ns, Qs),\ \underline{\mathit{safequeen\_list}([Q], Qs)},\ \mathit{safeboard}(Qs)$

which is the body of clause 16 and the goal

$\mathit{placequeens}(Ns1, Qs),\ \underline{\mathit{safequeen\_list}([Q, Q1], Qs)},\ \mathit{safeboard}(Qs)$

occurring in the body of clause 17. Then, by folding clause 16 (which was derived from clause 11 by replacing the atom *safequeen* by the corresponding atom *safequeen_list*) we get:

19. $t1(Q, Ns, Qs) \leftarrow \mathit{gent}(Ns, Qs, Q, [\ ])$

The program derived so far is made out of clauses 8.f, 9, 10.f, 12, 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 for which we will need a second application of the list introduction strategy.

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

20. $\mathit{gent}([\ ], [\ ], Q, Ps) \leftarrow \mathit{safequeen\_list}([Q|Ps], [\ ]),\ \mathit{safeboard}([\ ])$

21. $\mathit{gent}(Ns, [Q1|Qs], Q, Ps) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \mathit{safequeen\_list}([Q|Ps], [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

Then we unfold clause 20 w.r.t. *safeboard* and since $\mathit{safequeen\_list}([Q|Ps], [\ ]) \leftrightarrow \mathit{true}$ in the least Herbrand model of the program at hand (see clauses 14 and 15), by applying the goal replacement rule we get:

20.1 $\mathit{gent}([\ ], [\ ], Q, Ps) \leftarrow$

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

22. $\mathit{gent}(Ns, [Q1|Qs], Q, Ps) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \mathit{notattack}(Q, Q1),\ \underline{\mathit{safequeen}(Q, Qs)},$
$\qquad \mathit{safequeen\_list}(Ps, [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

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

23. $\mathit{gent}(Ns, [Q1|Qs], Q, [\ ]) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \mathit{notattack}(Q, Q1),\ \mathit{safequeen}(Q, Qs),\ \mathit{safeboard}([Q1|Qs])$

24. $\mathit{gent}(Ns, [Q1|Qs], Q, [Q2|Ps]) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \mathit{notattack}(Q, Q1),$
$\qquad \mathit{safequeen}(Q, Qs), \mathit{safequeen}(Q2, [Q1|Qs]),$
$\qquad \mathit{safequeen\_list}(Ps, [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

By unfolding clause 24 w.r.t. $\mathit{safequeen}(Q2, [Q1|Qs])$ we have:

25. $\mathit{gent}(Ns, [Q1|Qs], Q, [Q2|Ps]) \leftarrow \mathit{select}(Q1, Ns, Ns1),\ \mathit{placequeens}(Ns1, Qs),$
$\qquad \mathit{notattack}(Q, Q1),\ \mathit{notattack}(Q2, Q1),$
$\qquad \underline{\mathit{safequeen}(Q, Qs)},\ \underline{\mathit{safequeen}(Q2, Qs)},$
$\qquad \mathit{safequeen\_list}(Ps, [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

Let us now look at the unfolding process from clause 22 to clause 25. Similarly to the unfolding steps which led us 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 for a second time the list introduction strategy. We proceed as follows.

By applying the goal replacement rule we replace the underlined atoms of clauses 22 and 25 by the corresponding *safequeen_list* atoms, and we get:

$$
\begin{aligned}
26.\quad & gent(Ns,[Q1|Qs],Q,Ps) \leftarrow select(Q1,Ns,Ns1),\ placequeens(Ns1,Qs),\ notattack(Q,Q1),\\
& \qquad \underline{safequeen\_list([Q],Qs)}, safequeen\_list(Ps,[Q1|Qs]),\\
& \qquad safeboard([Q1|Qs])
\end{aligned}
$$

$$
\begin{aligned}
27.\quad & gent(Ns,[Q1|Qs],Q,[Q2|Ps]) \leftarrow select(Q1,Ns,Ns1),\ placequeens(Ns1,Qs),\\
& \qquad notattack(Q,Q1),\ notattack(Q2,Q1),\\
& \qquad \underline{safequeen\_list([Q,Q2],Qs)}, safequeen\_list(Ps,[Q1|Qs]),\\
& \qquad safeboard([Q1|Qs])
\end{aligned}
$$

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

$$
\begin{aligned}
28.\quad & ggent(Ns,Qs,Q,Ps,Ps1,Q1) \leftarrow placequeens(Ns,Qs),\\
& \qquad safequeen\_list([Q|Ps],Qs),\ safequeen\_list(Ps1,[Q1|Qs]),\\
& \qquad safeboard([Q1|Qs])
\end{aligned}
$$

where body of this clause is a most specific generalization of the following two goals E and H which are the conjunctions of *all* atoms which depend on the variable Qs and occur in the body of clauses 26 and 27, respectively:

$$
\begin{aligned}
E.\quad & placequeens(Ns1,Qs),\ safequeen\_list([Q],Qs),\ safequeen\_list(Ps,[Q1|Qs]),\\
& safeboard([Q1|Qs])\\
H.\quad & placequeens(Ns1,Qs),\ safequeen\_list([Q,Q2],Qs),\ safequeen\_list(Ps,[Q1|Qs]),\\
& safeboard([Q1|Qs])
\end{aligned}
$$

Since the constraint ($\tau 1$) is satisfied, we can then fold clause 26 (which was derived from clause 22 by replacing the atom *safequeen* by the corresponding atom *safequeen_list*) using clause 28. We get:

$$
\begin{aligned}
26.\mathrm{f}\quad & gent(Ns,[Q1|Qs],Q,Ps) \leftarrow select(Q1,Ns,Ns1),\ notattack(Q,Q1),\\
& \qquad ggent(Ns1,Qs,Q,[\,],Ps,Q1)
\end{aligned}
$$

The program derived so far is made out of clauses 8.f, 9, 10.f, 12, 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 the recursive definition of this predicate. The corresponding derivation is shown in Appendix A. It is a simple derivation which requires unfolding and folding steps only. The final program we get is as follows:

n-queens program with accumulators

$$
\begin{aligned}
8.\mathrm{f}\quad & queens(Ns,Qs) \leftarrow t(Ns,Qs)\\
9.\quad & t([\,],[\,]) \leftarrow\\
10.\mathrm{f}\quad & t(Ns,[Q|Qs]) \leftarrow select(Q,Ns,Ns1),\ t1(Q,Ns1,Qs)\\
12.\quad & t1(Q,[\,],[\,]) \leftarrow\\
19.\quad & t1(Q,Ns,Qs) \leftarrow gent(Ns,Qs,Q,[\,])
\end{aligned}
$$

20.1 $gent([\,],[\,],Q,Ps) \leftarrow$
26.f $gent(Ns,[Q1|Qs],Q,Ps) \leftarrow select(Q1,Ns,Ns1),\ notattack(Q,Q1),$
$\qquad ggent(Ns1,Qs,Q,[\,],Ps,Q1)$
29.3 $ggent(Ns,Qs,Q,Ps,[\,],Q1) \leftarrow gent(Ns,Qs,Q1,[Q|Ps])$
30.3 $ggent(Ns,Qs,Q,Ps,[Q2|Ps1],Q1) \leftarrow notattack(Q2,Q1),$
$\qquad ggent(Ns,Qs,Q2,\ [Q|Ps],Ps1,Q1)$

This program performs much less backtracking than the initial program and its operational behaviour is similar to the *accumulator program* given in [16] at page 255. By clause 19 for predicate $t1$, the first queen position Q 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 which corresponds to the third argument of *gent*. When a new queen is placed at position $Q1$ and it is not attacked by a queen in position Q, the predicate *ggent* checks whether or not this queen is attacked by previously placed queens whose positions are in the list 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.

Using SICStus Prolog on a Sparc Ultra the initial program finds all solutions for 8 queens in 18.5 seconds, while the final program takes 1.1 seconds.

As a last remark, notice that we can simplify the final program we have derived by performing unfolding steps w.r.t. the atoms with the non-recursive predicates t and $t1$. 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 conjunction (or a list) of goals by a single atom having a list of values as one of its arguments. Let us first consider two special cases of application of this strategy.

In the first special case we replace a conjunction 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 predicate *p_list* is defined by the clauses $L1$ and $L2$ (see end of Section 1) which we rewrite here for the reader's convenience:

$L1.$ $p\_list([\,],Y) \leftarrow$
$L2.$ $p\_list([X|Xs],Y) \leftarrow p(X,Y),\ p\_list(Xs,Y)$

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 conjunction of atoms, such as Y in the conjunction (1), occur at the same argument position.

This motivates the following second special case of the list introduction strategy. Let us consider the following conjunction 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. We can avoid the intermediate variables $T_1, \ldots, T_{n-1}$ by replacing the given conjunction (2) by the single atom $q\_list(T_0, [X_1, \ldots, X_n], T_n)$, defined as follows:

$M1.\quad q\_list(T, [\,], T) \leftarrow$

$M2.\quad q\_list(T_0, [X|Xs], T_n) \leftarrow q(T_0, X, T_1),\; q\_list(T_1, Xs, T_n)$

The use of the predicate *q_list* corresponds to the use of the higher-order function *foldl*, which is defined as we now recall. Given a set S and a binary function $\oplus$ over that set, we have that:

$foldl\ \oplus\ T\ [\,] = T$ for any element T in S,

$foldl\ \oplus\ T_0\ (L@[X_n]) = ((foldl\ \oplus\ T_0\ L) \oplus X_n)$ for any list L of elements in S and any pair of elements T_0 and X_n in S (here @ denotes the *append* function on lists).

Now the correspondence between *q_list* and *foldl* can easily be established. Indeed, if we assume that $q(T, X, T')$ holds iff $T \oplus X = T'$, we have that:

$q\_list(T_0, [X_1, \ldots, X_n], T_n)$ iff $\quad$ {by clauses $M1$ and $M2$}

$q(T_0, X_1, T_1),\ q(T_1, X_2, T_2),\ \ldots,\ q(T_{n-1}, X_n, T_n)$ iff $\quad$ {by definition of q}

$(\ldots((T_0 \oplus X_1) \oplus X_2) \ldots \oplus X_n)$ iff $\quad$ {by definition of *foldl*}

$foldl\ \oplus\ T_0\ [X_1, \ldots, X_n] = T_n$.

Now for any given predicate p we provide the general definition of the corresponding predicate *p_list* which in particular, works for conjunctions of atoms of the form (1) and (2) shown above. In order to present the list introduction strategy in the general case where we allow for the replacement of a conjunction of possibly *non-atomic* goals by a single atom, we need to introduce the following notation and terminology.

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.

By $\overline{X}$ we denote a tuple of elements of the form $\langle X_1, \ldots, X_n \rangle$ whose length n (≥ 0) is left unspecified. The tuple for $n = 0$, called the empty tuple, is denoted by ε. Two tuples are said to be *disjoint* iff no element occurs in both of them.

The sequence $(\overline{V}_1, \ldots, \overline{V}_k)$ of tuples is said to be a *partition* of the set V iff (i) each tuple has no repeated elements and (ii) every element of V occurs in exactly one $\overline{V}_i$ for some i in $\{1, \ldots, k\}$.

In order to denote goals and to identify the variables occurring in them, we use greek letters having partitions of sets of variables as arguments. For instance, $\pi(\overline{V}_1, \ldots, \overline{V}_k)$ denotes a goal such that $(\overline{V}_1, \ldots, \overline{V}_k)$ is a partition of the set of variables occurring in that goal.

Let us consider a goal $\pi(\overline{V}_1, \ldots, \overline{V}_k)$ and a sequence $(\overline{W}_1, \ldots, \overline{W}_k)$ of k tuples of variables such that $\overline{W}_i$ has the same length of $\overline{V}_i$, for $i = 1, \ldots, k$. Then, when we write $\pi(\overline{V}_1/\overline{W}_1, \ldots, \overline{V}_k/\overline{W}_k)$, or simply $\pi(\overline{W}_1, \ldots, \overline{W}_k)$, we denote the goal which is obtained from $\pi(\overline{V}_1, \ldots, \overline{V}_k)$ by replacing every variable in the tuple $\overline{V}_i$ by the corresponding variable in the tuple $\overline{W}_i$, for $i = 1, \ldots, k$.

Definition 1. *A goal R occurring in the body of a clause C is said to be a* goal list of length n based on the goal $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T'}, \overline{Y})$ *iff R is a conjunction of n (≥ 0) goals of the form:*

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

such that: (i) each of those goals is a variant *of $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T'}, \overline{Y})$, (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, possibly empty tuples of variables, (iii) for $i = 1, \ldots, n$, $\overline{I}_i$ is a tuple of internal variables of the goal $\pi(\overline{I}_i, \overline{T}_{i-1}, \overline{X}_i, \overline{T}_i, \overline{Y})$ in C, and (iv) $\overline{T}_1, \ldots, \overline{T}_{n-1}$ are tuples of internal variables of R in C.*

The following clause D clarifies the above definition of a goal list:

$$\begin{array}{lll} D. & h(U_0,V_0,U_3,V_3,X_1,Y) \leftarrow & r(U_0,U_1),\ s(V_1,Y,V_0),\ t(I_1,X_1),\\ & & r(U_1,U_2),\ s(V_2,Y,V_1),\ t(I_2,X_2),\\ & & r(U_2,U_3),\ s(V_3,Y,V_2),\ t(I_3,X_3) \end{array}$$

The body of D is a goal list of length 3 based on the goal $r(U,U'),\ s(V',Y,V),\ t(I,X)$ which can be denoted by $\delta(\langle I\rangle,\langle U,V\rangle,\langle X\rangle,\langle U',V'\rangle,\langle Y\rangle)$.

The following definition introduces 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:*

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

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.

As an example of Definition 2, here are the two clauses which define the predicate *d_list* which is the list generalization of the goal $\delta(\langle I\rangle,\langle U,V\rangle,\langle X\rangle,\langle U',V'\rangle,\langle Y\rangle)$ considered above:

$$\begin{array}{lll} D1. & d\_list(\langle U,V\rangle,[\ ],\langle U,V\rangle,\langle Y\rangle) \leftarrow & \\ D2. & d\_list(\langle U_0,V_0\rangle,[\langle X\rangle|Xs],\langle U_n,V_n\rangle,\langle Y\rangle) \leftarrow & r(U_0,U_1),\ s(V_1,Y,V_0),\ t(I,X),\\ & & d\_list(\langle U_1,V_1\rangle,Xs,\langle U_n,V_n\rangle,\langle Y\rangle) \end{array}$$

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:

$$\begin{array}{ll} M(P) \models \forall \overline{T}_0,\overline{X}_1,\ldots,\overline{X}_n,\overline{T}_n,\overline{Y}.\ ((\exists \overline{I}_1,\ldots,\overline{I}_n,\overline{T}_1,\ldots,\overline{T}_{n-1}.\ R) \leftrightarrow & \\ \qquad\qquad p\_list(\overline{T}_0,[\overline{X}_1,\ldots,\overline{X}_n],\overline{T}_n,\overline{Y})) & \square \end{array}$$

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

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

$$\begin{array}{l} M(P) \models \forall \overline{T}_0,L_1,L_2,\overline{T}_n,\overline{Y}.\ \ (\exists \overline{T}.\ (p\_list(\overline{T}_0,L_1,\overline{T},\overline{Y}),\ p\_list(\overline{T},L_2,\overline{T}_n,\overline{Y})) \leftrightarrow \\ \qquad\qquad \exists L.\ (append(L_1,L_2,L),\ p\_list(\overline{T}_0,L,\overline{T}_n,\overline{Y}))) \end{array}$$

where *append* is the predicate which specifies list concatenation. $\square$

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 [3].

Let us now describe the list introduction strategy. This strategy combined with the tupling strategy can be used in the case when a variable number of calls should be tupled together to allow for a folding step which collects *all* calls of a given predicate in a clause. The application of the list introduction strategy essentially consists in: (i) matching a given clause, say U, against 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*, (iii) introducing a new predicate defined by a clause whose body is a most specific generalization of goals occurring in the bodies of *U_list* and *V_list*, and finally (iv) looking for the recursive definition of the newly introduced predicate, by performing unfolding, folding, and goal replacement steps, possibly based on Properties P1 and P2.

We first introduce the following concept.

Definition 3. *Let us consider two goal lists R and S of length r and s, respectively, based on the same goal. We say that the goal list S is an* extension *of the goal list R iff $S = (R, W)$ for some goal list W with positive length.*

The List Introduction Strategy

As we have seen in the Introduction, during program derivation when applying the tupling or the generalization strategy, we may introduce new clauses for the definition of new predicates. Let us consider one of those clauses, say C, and suppose that by performing some unfolding and/or goal replacement steps, from clause C we derive a clause of the form (recall that the order of the atoms is immaterial):

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

where the E_i's and the H_i's are atoms and F is a goal list. Suppose also that from U, by further applications of the unfolding and/or goal replacement rules not involving the atoms $H_1, \ldots, H_t$, we derive the following clause:

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

where the I_i's and the M_i's are atoms and L is a goal list, and (1) for $i = 1, \ldots, r$, the atom I_i has the same predicate symbol of the atom E_i, and (2) F and L are goal lists based on the same goal, say $\pi(\overline{I}, \overline{T}, \overline{X}, \overline{T'}, \overline{Y})$, such that L is an extension of F.

The list introduction strategy consists of the following five steps.

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 $C2$ given in Definition 2.

Step 2. By applying Property P1 we replace the goal lists in the clauses U and V by *p_list* atoms and we get the following two clause, respectively:

$U\_list.\quad P \leftarrow E_1, \ldots, E_r,\ p\_list(\overline{T}_0, \ell_F, \overline{T}_n, \overline{Y}),\ H_1, \ldots, H_t$

$V\_list.\quad Q \leftarrow I_1, \ldots, I_r,\ p\_list(\overline{T}_0, \ell_L, \overline{T}_m, \overline{Y}),\ M_1, \ldots, M_d$

The values of the list arguments ℓ_F and ℓ_L are the ones which make Property P1 true. 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:

$G.\quad genp\_list(V_1, \ldots, V_k) \leftarrow N_1, \ldots, N_r,\ p\_list(\overline{T}_0, \ell, \overline{T}, \overline{Y})$

whose body is a most specific generalization of the goal

$E_1, \ldots, E_r,\ p\_list(\overline{T}_0, \ell_F, \overline{T}_n, \overline{Y})$

occurring in the body of *U_list* and the goal

$I_1, \ldots, I_r,\ p\_list(\overline{T}_0, \ell_L, \overline{T}_m, \overline{Y})$

occurring in the body of *V_list*. The variables $V_1, \ldots, V_k$ are chosen as we have 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 a suitable combination of the tupling, generalization, and list introduction strategies. 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 mentioned in the above description of 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 the Introduction 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 seen that the application of the tupling strategy and the generalization 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 from two to three when performing the sequence of unfolding steps which lead us from clause 4 to clause 7.

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 *generalization + equality introduction* rule [14] which as we will apply it in our derivation, is an instance of the goal replacement rule. The generalization + equality introduction rule 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 two 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 $\mu(\varepsilon, \langle J1, I\rangle, \langle K1\rangle, \langle J, I1\rangle, \varepsilon)$ 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 $\mu(\varepsilon, \langle J1, I\rangle, \langle K1\rangle, \langle J, I1\rangle, \varepsilon)$. Actually, F and (L_1, L_2) are goal lists based on $\mu(\varepsilon, \langle J1, I\rangle, \langle K1\rangle, \langle J, I1\rangle, \varepsilon)$. 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. $m\_list(J, I, [\ ], J, I) \leftarrow$
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, we have dropped arguments with empty tuples, and we have identified the singleton tuple $\langle K1\rangle$ with the variable $K1$.

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. $genm\_list(I, K1, Ks, J, K2) \leftarrow m\_list(J1, I, [K1|Ks], J, I1),\ p(I1, J, K2)$

whose body is a 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. In this generalization step we have considered the atoms with predicates *m_list* and p, instead of *m_list* only, because as already mentioned in the Introduction, we look for a linear recursive program and *m_list* depends on p.

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

15. $tp(I, J, K1, K2) \leftarrow genm\_list(I, K1, [\,], J, K2)$

Step 5. We are now left with the task of deriving the recursive definition of *genm_list*. This derivation is shown in Appendix B and it is performed by applying again the list introduction and generalization strategies. 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. $tp(I, J, K1, K2) \leftarrow genm\_list(I, K1, [\,], J, K2)$
19. $genm\_list(A, B, C, D, E) \leftarrow new1(A, [B|C], D, E)$
20. $new1(A, [\,], B, C) \leftarrow p(A, B, C)$

21.f $new1(0, [1|A], B, C) \leftarrow new1(s(0), A, B, C)$
22.f $new1(s(A), [B|C], D, E) \leftarrow new2(A, G, [\,], H, C, D, E),\ ave(G, H, B)$
25.f $new2(A, B, C, D, [\,], 0, 0) \leftarrow genm\_list(A, B, C, 0, D)$
26.f $new2(A, B, C, D, [\,], s(E), F) \leftarrow append(C, [D], G),\ genm\_list(A, B, G, E, H),\ ave(D, H, F)$
27.f $new2(A, B, C, D, [E|F], G, H) \leftarrow append(C, [D], I),\ new2(A, B, I, J, F, G, H),\ ave(D, J, E)$

This derived program is linear recursive and it requires $O(m \times n)$ calls of *ave* for evaluating a goal of the form $p(s^m(0), s^n(0), K)$. Our final program could be simplified by unfolding the calls of *tp* and *genm_list* in the derived clause. We may then discard the clauses which define *tp* and *genm_list* (that is, clauses 15 and 19) because they will no longer be necessary 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 [22].

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 [5]. 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 conjunctions of predicate calls.

A related idea of encoding conjunctions of goals by lists of values was also used in the compiling control transformation technique proposed in [4]. 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 conjunction 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 [14].

Our transformational approach also contrasts with other approaches considered in the case of functional languages for solving similar problems such as those described in [8] and in [7], 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 conjunctions 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). In this case our transformation technique is related to the *continuation passing style transformation* proposed in [21] for functional programs. Continuation passing style transformations have also been proposed for logic programs [15, 18]. Those transformations are realized by encoding the stack of suspended goals as a term. However, by themselves those transformations do not improve program efficiency, whereas 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 definition at the beginning of the transformation process, (ii) the atoms to unfold, (iii) the two clauses to be matched for the application of the generalization strategy or the list introduction strategy, (iv) the conjunctions of goals to be replaced by a single atom when applying list introduction, and finally, (v) when and how to perform some goal replacement steps such as the generalization + equality introduction steps in the World Series Odds example.

There are no general and powerful techniques for addressing all these issues in a completely satisfactory way, and for the time being we can only offer partial answers.

For Point (i) one suitable suggestion is to tuple in the new definition the atoms which have common arguments or have arguments which share 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. This allows us 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 definition to be introduced may be decided after analyzing the program at hand (see, for instance, the work by Cohen [8]).

For Point (ii) and Point (iii) much work has been done in the framework of several related techniques, such as *supercompilation* [19], *deforestation* [20], the automation of the tupling strategy [6], *partial evaluation* [11], 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. In particular, in the derivation of the *n-queens* program some of the unfolding steps were performed according to the Synchronized Descend Rule [14].

For Point (iv) we would like to notice that, for a given clause, there may be several ways, possibly conflicting ones, of arranging a conjunction of atoms in its body as a goal list according to Definition 1. As a consequence, there may be several ways of applying the list introduction strategy. It is hard to define general policies to choose among these conflicting options so to get the best final program. However, in practice, it is often relatively 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.

Finally, for Point (v) we notice that the application of the generalization + equality introduction rule, as well as other goal replacements, may be guided by the need for generating two goal lists, say R and S, such that S is an extension of R, thereby significantly restricting the set of possible replacements that can be made.

Appendix A

Derivation of the recursive definition of the predicate *ggent* defined by clause 28 in the *n-queens* example.

By unfolding clause 28 w.r.t. $\mathit{safequeen\_list}(Ps1, [Q1|Qs])$ we get clauses 29 and 30:

29. $\mathit{ggent}(Ns, Qs, Q, Ps, [\,], Q1) \leftarrow \mathit{placequeens}(Ns, Qs), \mathit{safequeen\_list}([Q|Ps], Qs),$
$\mathit{safeboard}([Q1|Qs])$

30. $\mathit{ggent}(Ns, Qs, Q, Ps, [Q2|Ps1], Q1) \leftarrow \mathit{placequeens}(Ns, Qs),\ \mathit{safequeen\_list}([Q|Ps], Qs),$
$\mathit{safequeen}(Q2, [Q1|Qs]),\ \mathit{safequeen\_list}(Ps1, [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

By unfolding clause 29 w.r.t. $\mathit{safeboard}([Q1|Qs])$ we get:

29.1 $\mathit{ggent}(Ns, Qs, Q, Ps, [\,], Q1) \leftarrow \mathit{placequeens}(Ns, Qs),$
$\underline{\mathit{safequeen\_list}([Q|Ps], Qs)},\ \underline{\mathit{safequeen}(Q1, Qs)},\ \mathit{safeboard}(Qs)$

By folding clause 29.1 w.r.t. the underlined atoms using clause 15, we get:

29.2 $\mathit{ggent}(Ns, Qs, Q, Ps, [\,], Q1) \leftarrow \mathit{placequeens}(Ns, Qs),$
$\mathit{safequeen\_list}([Q1, Q|Ps], Qs),\ \mathit{safeboard}(Qs)$

By folding clause 29.2 w.r.t. the whole body using clause 18, we get:

29.3 $\mathit{ggent}(Ns, Qs, Q, Ps, [\,], Q1) \leftarrow \mathit{gent}(Ns, Qs, Q1, [Q|Ps])$

By unfolding clause 30 w.r.t. $\mathit{safequeen}(Q2, [Q1|Qs])$:

30.1 $\mathit{ggent}(Ns, Qs, Q, Ps, [Q2|Ps1], Q1) \leftarrow \mathit{placequeens}(Ns, Qs),$
$\underline{\mathit{safequeen\_list}([Q|Ps], Qs)},\ \mathit{notattack}(Q2, Q1),$
$\underline{\mathit{safequeen}(Q2, Qs)},\ \mathit{safequeen\_list}(Ps1, [Q1|Qs]),\ \mathit{safeboard}([Q1|Qs])$

By folding clause 30.1 w.r.t. the underlined atoms using clause 15, we get:

30.2 $\mathit{ggent}(Ns, Qs, Q, Ps, [Q2|Ps1], Q1) \leftarrow \underline{\mathit{placequeens}(Ns, Qs)},\ \mathit{notattack}(Q2, Q1),$
$\underline{\mathit{safequeen\_list}([Q2, Q|Ps], Qs)},\ \underline{\mathit{safequeen\_list}(Ps1, [Q1|Qs])},\ \underline{\mathit{safeboard}([Q1|Qs])}$

By folding clause 30.2 w.r.t. the underlined atoms using clause 28, we get:

30.3 $\mathit{ggent}(Ns, Qs, Q, Ps, [Q2|Ps1], Q1) \leftarrow \mathit{notattack}(Q2, Q1),$
$\mathit{ggent}(Ns, Qs, Q2, [Q|Ps], Ps1, Q1).$

Appendix B

Derivation of the recursive definition of the predicate *genm_list* defined by clause 14 in the World Series Odds example.

We start from clause 14:

14. $genm\_list(I, K1, Ks, J, K2) \leftarrow m\_list(J1, I, [K1|Ks], J, I1),\ p(I1, J, K2)$

By unfolding this clause w.r.t. *m_list* and by unfolding the resulting clause w.r.t. the generated equalities and the generated atom with predicate p, we get:

16. $genm\_list(0, 1, A, B, C) \leftarrow m\_list(D, s(0), A, B, E),\ p(E, B, C)$

17. $genm\_list(s(A), B, C, D, E) \leftarrow p(A, s(F), G),\ p(s(A), F, H),\ ave(G, H, B),$
$m\_list(F, s(s(A)), C, D, I),\ p(I, D, E).$

We apply the generalization strategy and by matching the bodies of clauses 14 and 16 we introduce the new predicate $new1$ defined by the clause:

18. $new1(A, B, C, D) \leftarrow m\_list(E, A, B, C, F),\ p(F, C, D)$

whose body is a most specific generalization of the bodies of clauses 14 and 16. We then fold clause 14 using clause 18 and we get:

19. $genm\_list(A, B, C, D, E) \leftarrow new1(A, [B|C], D, E)$

Now we look for the recursive definition of the predicate $new1$. Starting from clause 18, after some unfolding steps w.r.t. *m_list*, p, and the generated equalities, we get the following clauses:

20. $new1(A, [\,], B, C) \leftarrow p(A, B, C)$

21. $new1(0, [1|A], B, C) \leftarrow m\_list(D, s(0), A, B, E),\ p(E, B, C)$

22. $new1(s(A), [B|C], D, E) \leftarrow p(A, s(F1), G),\ p(s(A), F1, H),\ ave(G, H, B),$
$m\_list(F1, s(s(A)), C, D, J),\ p(J, D, E)$

We then fold clause 21 using clause 18 and we get:

21.f $new1(0, [1|A], B, C) \leftarrow new1(s(0), A, B, C)$

We are left with the problem of finding a linear recursive program equivalent to clause 22. We first apply the generalization + equality introduction rule and we transform clause 22 into the following clause:

22.g $new1(s(A), [B|C], D, E) \leftarrow \underline{F2 = s(F1),\ p(A, F2, G),\ A1 = s(A)},\ p(A1, F1, H),$
$ave(G, H, B),\ m\_list(F1, s(A1), C, D, J),\ p(J, D, E)$

We then apply the list introduction strategy. Clauses U, V, *U_list*, *V_list*, and G which we introduced when describing this strategy in Section 3, correspond, respectively, to the clauses 22.g, 23.g, 22.1, 23.1, and 24 (see below).
Starting from clause 22.g, after a few unfolding steps w.r.t. *m_list*, the equalities, and the predicate p, we use the functionality of p and we get the following two clauses:

$new1(s(A), [B], C, D) \leftarrow p(A, s(C), E),\ p(s(A), C, F),\ ave(E, F, B),\ p(s(s(A)), C, D)$

23. $new1(s(A), [B1, B2|C], D, E) \leftarrow p(A, s(s(F)), G),\ p(s(A), s(F), H),$
$ave(G, H, B1),\ p(s(s(A)), F, I),\ ave(H, I, B2),$
$m\_list(F, s(s(s(A))), C, D, J),\ p(J, D, E)$

By applying the generalization + equality introduction rule from clause 23 we get:

$$23.g \quad new1(s(A),[B1,B2|C],D,E) \leftarrow \underline{F2{=}s(F1),\ p(A,F2,G),\ A1{=}s(A),} \\ \underline{F1{=}s(F),\ p(A1,F1,H),\ A2{=}s(A1)}, \\ ave(G,H,B1),\ p(A2,F,I),\ ave(H,I,B2), \\ m\_list(F,s(A2),C,D,J),\ p(J,D,E)$$

The underlined goal in the body of clause 23.g is an extension of the underlined goal in the body of clause 22.g. Thus, we perform the five steps of the list introduction strategy described in Section 3 as follows.

Steps 1 and 2.
We consider the list generalization of the goal $F2{=}s(F1),\ p(A,F2,G),\ A1{=}s(A)$, that is, the predicate *m_list* defined by clauses 10 and 11, and we transform the underlined goal in clause 22.g by using Property P1. We get:

$$22.1 \quad new1(s(A),[B|C],D,E) \leftarrow \underline{m\_list(F2,A,[G],F1,A1),\ p(A1,F1,H)}, \\ ave(G,H,B),\ m\_list(F1,s(A1),C,D,J),\ p(J,D,E)$$

Similarly, we transform the underlined goal in clause 23.g by using Property P1 and we get:

$$23.1 \quad new1(s(A),[B1,B2|C],D,E) \leftarrow \underline{m\_list(F2,A,[G,H],F,A2)}, \\ ave(G,H,B1),\ p(A2,F,I),\ ave(H,I,B2), \\ m\_list(F,s(A2),C,D,J),\ p(J,D,E)$$

Step 3. By matching clauses 22.1 and 23.1 we apply the generalization strategy and we introduce the new predicate *new*2 defined by the clause:

$$24. \quad new2(A,G,Gs,I,C,D,E) \leftarrow m\_list(F2,A,[G|Gs],F,A2),\ p(A2,F,I), \\ m\_list(F,s(A2),C,D,J),\ p(J,D,E)$$

whose body is a most specific generalization of the bodies of clauses 22.1 and 23.1, discarding the atoms with predicate *ave*.

Step 4. We fold clause 22.1 using clause 24 and we get:

$$22.f \quad new1(s(A),[B|C],D,E) \leftarrow new2(A,G,[\ ],H,C,D,E),\ ave(G,H,B)$$

Step 5. Now we look for the recursive clauses defining *new*2. By unfolding clause 24 w.r.t. $m\_list(F,s(A2),C,D,J)$ and then the resulting clauses w.r.t. the generated equalities and the atom which is a variant of $p(s(A),B,C)$, we get:

$$25. \quad new2(A,B,C,D,[\ ],0,0) \leftarrow m\_list(E,A,[B|C],0,F),\ p(F,0,D)$$
$$26. \quad new2(A,B,C,D,[\ ],s(E),F) \leftarrow m\_list(G,A,[B|C],s(E),H), \\ \underline{p(H,s(E),D)},\ \underline{p(H,s(E),I)},\ p(s(H),E,J),\ ave(I,J,F)$$
$$27. \quad new2(A,B,C,D,[E|F],G,H) \leftarrow m\_list(I,A,[B|C],s(J),K),\ p(K,s(J),D), \\ p(s(K),s(J),E),\ m\_list(J,s(s(K)),F,G,L),\ p(L,G,H)$$

We fold clause 25 using clause 14 and we get:

$$25.f \quad new2(A,B,C,D,[\ ],0,0) \leftarrow genm\_list(A,B,C,0,D)$$

Starting from clause 26 we use the functionality of the predicate p (see the underlined atoms), we apply the generalization + equality introduction rule, and we apply Property P1. We get the clause:

$$26.1 \quad new2(A,B,C,D,[\ ],s(E),F) \leftarrow \underline{m\_list(G,A,[B|C],H,I)},\ \underline{m\_list(H,I,[D],E,J)}, \\ p(J,E,K),\ ave(D,K,F)$$

Then we apply Property P2 w.r.t. the underlined atoms and we unfold the *append* atom introduced by Property P2. We can fold the resulting clause using clause 14 and we get:

26.f $new2(A,B,C,D,[\,],s(E),F) \leftarrow append(C,[D],G),\ genm\_list(A,B,G,E,H),$
$ave(D,H,F)$

Finally, starting from clause 27 we unfold it w.r.t. the atom $p(s(K),s(J),E)$, we use the functionality of the predicate p, we apply the generalization + equality introduction rule, and we apply Property P1. We get the clause:

27.1 $new2(A,B,C,D,[E|F],G,H) \leftarrow \underline{m\_list(I,A,[B|C],J,K)},\ \underline{m\_list(J,K,[D],L,M)},$
$p(M,L,N),\ ave(D,N,E),\ m\_list(L,s(M),F,G,P),\ p(P,G,H)$

Then we apply Property P2 w.r.t. the underlined atoms and we unfold the *append* atom introduced by Property P2. We can fold the resulting clause using clause 24 and we get:

27.f $new2(A,B,C,D,[E|F],G,H) \leftarrow append(C,[D],I),\ new2(A,B,I,J,F,G,H),$
$ave(D,J,E)$

Thus, the definition of *genm_list* is given by clauses 19, 20, 21.f, 22.f, 25.f, 26.f, and 27.f.

Acknowledgements

We would like to thank Danny De Schreye for many fruitful conversations and for comments on the n-queens example presented in this paper. Sophie Renault implemented an interactive program transformation system which helped us for the development of the examples. We also thank Anna Formica for reading an earlier draft of this paper.

References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] R. S. Bird, "The promotion and accumulation strategies in transformational programming," *ACM Toplas*, vol. 6, no. 4, pp. 487–504, 1984.

[3] R. S. Bird and L. G. L. T. Meertens, "Two exercises found in a book on algorithmics," in *TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, Germany* (L. G. L. T. Meertens, ed.), pp. 451–457, North-Holland, 1987.

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

[5] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, pp. 44–67, January 1977.

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

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

[8] N. H. Cohen, "Eliminating redundant recursive calls," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 265–299, 1983.

[9] J. Darlington, "A synthesis of several sorting algorithms," *Acta Informatica*, vol. 11, pp. 1–30, 1978.

[10] J. Darlington, "An experimental program transformation system," *Artificial Intelligence*, vol. 16, pp. 1–46, 1981.

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

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

[13] A. Pettorossi and M. Proietti, "Transformation of logic programs," in *Handbook of Logic in Artificial Intelligence and Logic Programming* (D. M. Gabbay, C. J. Hogger, and J. A. Robinson, eds.), vol. 5, pp. 697–787, Oxford University Press, 1998.

[14] M. Proietti and A. Pettorossi, "Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs," *Theoretical Computer Science*, vol. 142, no. 1, pp. 89–124, 1995.

[15] T. Sato and H. Tamaki, "Existential continuation," *New Generation Computing*, vol. 6, pp. 421–438, 1989.

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

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

[18] P. Tarau and M. Boyer, "Elementary logic programs," in *Proceedings PLILP '90* (P. Deransart and J. Małuszyński, eds.), pp. 159–173, Springer-Verlag, 1990.

[19] V. F. Turchin, "The concept of a supercompiler," *ACM TOPLAS*, vol. 8, no. 3, pp. 292–325, 1986.

[20] P. L. Wadler, "Deforestation: Transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, pp. 231–248, 1990.

[21] M. Wand, "Continuation-based program transformation strategies," *Journal of the ACM*, vol. 27, no. 1, pp. 164–180, 1980.

[22] J. Zhang and P. W. Grant, "An automatic difference-list transformation algorithm for Prolog," in *Proceedings 1988 European Conference on Artificial Intelligence, ECAI '88*, pp. 320–325, Pitman, 1988.