Formal Aspects of Computing (2002) 13: 233-251
(Special Issue in Honour of Rod Burstall).

The List Introduction Strategy for the Derivation of Logic Programs

Alberto Pettorossi[1] and Maurizio Proietti[2]

[1]Dipartimento di Informatica, Sistemi e Produzione,
Università di Roma Tor Vergata, Rome, Italy
[2]IASI-CNR, Rome, Italy

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 makes it possible to transform general recursive programs into linear recursive ones also in cases when this transformation cannot be performed by the tupling strategy. The linear recursive programs we derive by applying the list introduction strategy have in most cases very good time and space performance because they avoid repeated evaluations of goals and unnecessary constructions of data structures.

Keywords: Program transformation; Transformation rules and strategies; Logic programming; Program derivation; Automatic programming

1. Introduction

There are various methodologies for developing programs from specifications, and for deriving new, efficient programs from old programs. One such methodology is the *transformation methodology*, which has been first advocated by Burstall and Darlington [BuD77]. Using this methodology one can simplify or even avoid the proofs of program correctness. In particular, in the so called 'rules + strategies' approach it is not necessary to prove the correctness of the derived programs because the rules which are used for transforming programs, are guaranteed to preserve the semantics of interest.

The transformation methodology was first introduced in the case of functional languages, and it was later applied also to other language paradigms. Indeed, in the mid 1980s Tamaki and Sato [TaS84] applied the 'rules + strategies' approach to the transformation of logic programs, and more recently that approach has been applied by Seki to logic programs with negation in the bodies of the clauses [Sek91]. Different sets of transformation rules have been proposed in the literature for the different classes of programs which have been considered. The interested reader may refer to [PeP98] for a survey.

We will focus our attention on logic program transformation. This choice is motivated by the fact that in

Correspondence and offprint requests to: Maurizio Proietti, IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy. e-mail: proietti@iasi.rm.cnr.it

logic programming one can easily express non-deterministic computations as well as relational computations of the kind which naturally arise in many application areas, such as knowledge representation, databases and expert systems.

During program derivation the transformation rules are applied according to suitable strategies which have the objective of avoiding useless transformations and deriving very efficient programs. Here we want to consider, among the many strategies which have been proposed in the literature (see, for instance, [PeP96]), the so-called *tupling strategy*, whereby a collection of predicate calls are grouped together with the aim of avoiding common subcomputations and eliminating unnecessary data structures, and we will propose a new strategy for overcoming its limitations. The tupling strategy has been given particular attention since the beginning of the development of the program transformation methodology, because it can easily allow for efficiency improvements by making function (or predicate) calls interact. Through the use of arrays and fix-sized tables, the tupling strategy speeds up computations by memoing already computed values (see, for instance, [ChH95, Coh83, LiS99]). In particular, the introduction of arrays may turn recursive computations into iterative ones, and thus efficiency is improved because the overhead for stack manipulation is avoided.

Tupling has, however, some limitations which we will illustrate below, and in order to overcome them we will propose the *list introduction strategy*. It is well known that by introducing lists one can always avoid recursive computations in favour of iterative computations, but the naive application of this result does not improve program efficiency. The list introduction strategy, however, does improve efficiency, because it takes advantage of suitable properties of the lists which are introduced and, in particular, decreases their dimension and makes their manipulation faster.

The paper is structured as follows. In Section 2 we present our rules for transforming logic programs. We also present the tupling strategy and an example of its application. We illustrate its limitations using the Binomial Coefficients example. In Section 3 we formally define the list introduction strategy and we indicate how it can be combined with the tupling and generalisation strategies. In Section 4 we apply the list introduction strategy for deriving: (i) an efficient algorithm for the N-Queens problem which requires very little backtracking, and (ii) an algorithm for the World Series Odds problem which requires quadratic time, instead of exponential time. Finally, in Section 5 we compare our list introduction strategy with other techniques proposed in the literature and we also discuss how our strategy can be mechanised.

2. The Transformation Rules and the Tupling Strategy

In this section we describe the rules which we use for transforming logic programs. We also describe the tupling strategy and illustrate its limitations. We then motivate the use of the list introduction strategy as a way of overcoming those difficulties. We assume that the reader is familiar with the basic notions of logic programming as presented, for instance, in [Llo87].

2.1. The Transformation Rules

Logic programs are finite sets of definite clauses [Llo87] and from a logical point of view these sets denote conjunctions. Given a logic program P, by $M(P)$ we denote its least Herbrand model. Given a clause C we denote its head by $hd(C)$ and its body by $bd(C)$. By $R[S]$ we denote an expression where we single out an occurrence S of one of its subexpressions. By $R[\_]$ we denote the expression obtained from $R[S]$ by dropping the occurrence S. Given an expression E, by $vars(E)$ we denote the set of (free or bound) variables occurring in E. Given the expressions $E_1, \ldots, E_n$, we denote by $vars(E_1, \ldots, E_n)$ the set $vars(E_1) \cup \ldots \cup vars(E_n)$. Given a substitution θ of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, we denote by $vars(\theta)$ the set of variables $\{X_1, \ldots, X_n\} \cup vars(t_1, \ldots, t_n)$.

We say that C *is a clause for* p iff C is a clause of the form $p(\ldots) \leftarrow Body$. We say that a predicate p *depends on* a predicate q in program P iff *either* there exists in P a clause for p whose body contains an occurrence of q *or* there exists in P a predicate r such that p depends on r in P and r depends on q in P. We say that a predicate p depends on a clause C in program P iff C is a clause for predicate q and p depends on q in P.

Given a logic program P we assume that there exists a particular clause T of P, called *top level clause*. If T is a clause for predicate t, then t is said to be the *top level predicate* and it is understood that we intend

to use the program P to evaluate goals of the form $t(\ldots)$ only. We assume that no predicate in P depends on the top level predicate t and, in particular, t does not depend on itself.

The reader may refer to [Llo87] for other notions that we will use in the paper and that are not recalled here, such as unification, most general unifier (mgu), variant, renaming (or standardisation) apart, least Herbrand model, SLD-refutation and SLD-tree.

The program transformation process consists in constructing a sequence $P_0, \ldots, P_n$ of programs, called *transformation sequence*, starting from an initial program P_0. Let us assume that we have constructed the sequence $P_0, \ldots, P_k$ of programs. Then we may construct program P_{k+1} which is the next program in the sequence, by applying one of the following rules R1–R5. We denote by $Defs_k$ the set of all clauses which are introduced by the definition introduction rule (see rule R1 below) during the construction of $P_0, \ldots, P_k$. Thus, $Defs_0$ is empty.

R1. Definition Introduction. We introduce the following set of n (≥ 1) clauses, called *definitions*:

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

where: (i) *newp* is a new predicate symbol, that is, it does not occur in $P_0 \cup Defs_k$, and (ii) for $1 \leq i \leq h$, the variable X_i occurs in $Body_j$ for some j such that $1 \leq j \leq n$.
By this rule we derive the new program P_{k+1} and the new set $Defs_{k+1}$ of definitions by adding the above n clauses to the program P_k and to the set $Defs_k$, respectively.

Notice that: (i) we allow for the introduction of recursive definitions, and (ii) for $1 \leq j \leq n$, a variable occurring in $Body_j$ need not be in $\{X_1, \ldots, X_h\}$.

R2. Unfolding. Let C be a renamed apart clause in program P_k of the form: $H \leftarrow Body[A]$, where A is an atom. Let $E_1, \ldots, E_n$, with $n \geq 0$, be all the clauses of $P_0 \cup Defs_k$ such that for $i = 1, \ldots, n$, A is unifiable with the head of E_i with mgu θ_i. For $i = 1, \ldots, n$, let C_i be the clause $(H \leftarrow Body[bd(E_i)])\theta_i$. Then by unfolding C w.r.t. A using $E_1, \ldots, E_n$ we derive clauses $C_1, \ldots, C_n$. We derive program P_{k+1} by replacing C in P_k by $C_1, \ldots, C_n$.

Notice that an application of the unfolding rule to clause C of P_k when $n = 0$ amounts to the deletion of C from P_k.

R3. Folding. Let C be a clause in P_k of the form: $H \leftarrow Body[Q\,\theta]$, where Q is a goal. Let D be the only clause in $Defs_k$ for predicate *newp*. Suppose that: (1) D' is a variant of D of the form: $newp(X_1, \ldots, X_h) \leftarrow Q$, (2) $vars(D') \cap vars(H, Body[\_]) = \emptyset$, and (3) $vars(\theta) \cap vars(Q) \subseteq \{X_1, \ldots, X_h\}$.
By folding C w.r.t. $Q\,\theta$ using D we derive the clause E: $H \leftarrow Body[newp(X_1, \ldots, X_h)\,\theta]$. We derive program P_{k+1} by replacing C in P_k by E.

For instance, by folding clause C: $p(X) \leftarrow q(t(X), Y), r(Y)$ using the clause D: $a(U, V) \leftarrow q(t(U), V)$ via the substitution $\theta = \{U/X, V/Y\}$, we derive clause E: $p(X) \leftarrow a(X, Y), r(Y)$.

R4. Goal Replacement. Let C be a clause in P_k of the form $H \leftarrow Body[Q\,\theta]$, where Q is a goal. Let L be a closed formula, called a *replacement law*, of the form: $\forall X_1 \ldots X_u.\ (\exists Y_1 \ldots Y_v.\ Q \leftrightarrow \exists Z_1 \ldots Z_w.\ R)$, where $X_1, \ldots, X_u, Y_1, \ldots, Y_v, Z_1, \ldots, Z_w$ are distinct variables. Suppose that the following conditions hold:

(i) $vars(Q, R) \cap vars(H, Body[\_]) = \emptyset$;
(ii) $vars(\theta) \cap \{Y_1, \ldots, Y_v, Z_1, \ldots, Z_w\} = \emptyset$; and
(iii) $M(P_0 \cup Defs_k) \models L$.

By applying the replacement law L, from clause C we derive the clause E: $H \leftarrow Body[R\,\theta]$. From program P_k we derive the new program P_{k+1} by replacing C by E.

R5. Equality Introduction and Elimination. Let C be a clause of the form: $(H \leftarrow Body)\{X/t\}$, such that the variable X does not occur in t and let D be the clause: $H \leftarrow X{=}t,\ Body$.
By *equality introduction* we derive clause D from clause C, and from program P_k if C occurs in P_k, we derive the new program P_{k+1} by replacing C by D.
Analogously, by *equality elimination* we derive clause C from clause D, and from program P_k if D occurs in P_k, we derive program P_{k+1} by replacing D by C.

One can show that the above transformation rules R1–R5 are correct in the sense that they preserve the least Herbrand model as stated in the following Theorem 2.1.

We say that program P *existentially terminates* for a goal G, and we write $P{\downarrow}G$, iff for P and G *either* there exists an SLD-refutation *or* there exists a finite SLD-tree without SLD-refutations. A transformation sequence $P_0, \ldots, P_n$ is said to be *termination-preserving* iff for all ground atoms A, if $(P_0 \cup \mathit{Defs}_n){\downarrow}A$ then $P_n{\downarrow}A$.

Theorem 2.1 (Correctness of the transformation rules). Let $P_0, \ldots, P_n$ be a termination-preserving transformation sequence constructed by using the rules R1–R5. Then $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.

The following example shows that the termination preserving condition cannot be dropped.

Example 1. Let us consider the program P_0: $\{p \leftarrow q, \quad q \leftarrow\}$. Since $M(P_0) \models p \leftrightarrow q$, by goal replacement we get P_1: $\{p \leftarrow p, \quad q \leftarrow\}$. We have that $M(P_0) = \{p, q\} \neq M(P_1) = \{q\}$, and indeed, for the atom p program P_1 does not existentially terminate.

2.2. The Tupling Strategy

In this section we recall some basic facts about the *tupling strategy* for logic programs and we illustrate some limitations of this strategy through an example. The rest of the paper will be devoted to show that, by the list introduction strategy, we may overcome these limitations.

We start by presenting an example of application of the tupling strategy.

Example 2 (A Linear Recurrence Relation). Let us consider the following program defining a linear recurrence relation:

1. $t1(X, Y) \leftarrow p(X, Y)$
2. $p(0, 1) \leftarrow$
3. $p(1, 1) \leftarrow$
4. $p(2, 1) \leftarrow$
5. $p(X{+}3, Y3) \leftarrow p(X{+}2, Y2),\ p(X, Y),\ \mathit{plus}(Y2, Y, Y3)$

where $X + k$ stands for the term $s(s(\ldots(X)\ldots))$ with k occurrences of the successor function s and $\mathit{plus}(X, Y, Z)$ holds iff $X{+}Y{=}Z$. We assume that the top level clause is $t1$.

Since in the body of clause 5 there are two p calls, the program given above takes $O(2^k)$ resolution steps to evaluate goals of the form $t1(k, Y)$. We now apply the tupling strategy with the objective of avoiding an exponential number of redundant p calls and derive a linear time program. This strategy works by introducing new predicate definitions, often called *eureka definitions* following Burstall and Darlington [BuD77]. These definitions are clauses whose bodies consist of conjunctions of p calls. In our example we will introduce the following eureka definitions:

6. $t2(X, Y2, Y) \leftarrow p(X{+}2, Y2),\ p(X, Y)$
7. $t3(X, Y2, Y1, Y) \leftarrow p(X{+}2, Y2),\ p(X{+}1, Y1),\ p(X, Y)$

These two definitions are derived by applying the unfolding and goal replacement rules as we now describe. From the top level clause 1, by unfolding we get:

8. $t1(0, 1) \leftarrow$
9. $t1(1, 1) \leftarrow$
10. $t1(2, 1) \leftarrow$
11. $t1(X{+}3, Y3) \leftarrow p(X{+}2, Y2),\ p(X, Y),\ \mathit{plus}(Y2, Y, Y3)$

From the conjunction of the p atoms in the body of clause 11 we derive the eureka definition given by clause 6. We then unfold clause 6 and we get the following clauses (for reasons of readability we also rearrange the order of the atoms):

12. $t2(0, 1, 1) \leftarrow$
13. $t2(X{+}1, Y3, Y1) \leftarrow p(X{+}2, Y2),\ p(X{+}1, Y1),\ p(X, Y),\ \mathit{plus}(Y2, Y, Y3)$

Similarly to what we have done above starting from clause 11, from the conjunction of the p atoms in the body of clause 13 we derive the eureka definition given by clause 7. We then unfold clause 7 and we replace a goal of the form $p(X, Y),\ p(X, Y1)$ by $p(X, Y),\ Y{=}Y1$ (that is, we use the *functionality* of p). We derive the following clauses:

14. $t3(0, 1, 1, 1) \leftarrow$

15. $t3(X{+}1, Y2, Y1, Y) \leftarrow p(X{+}2, Y1),\ p(X{+}1, Y),\ p(X, Y4),\ plus(Y1, Y4, Y2)$

The 3-tuple of p atoms in the body of clause 15 is a variant of the body of clause 7. Thus, no new eureka definition is introduced, and we derive the following final program by folding clause 11 using clause 6, and by folding clauses 13 and 15 using clause 7:

8. $t1(0, 1) \leftarrow$
9. $t1(1, 1) \leftarrow$
10. $t1(2, 1) \leftarrow$
11f. $t1(X{+}3, Y3) \leftarrow t2(X, Y2, Y),\ plus(Y2, Y, Y3)$
12. $t2(0, 1, 1) \leftarrow$
13f. $t2(X{+}1, Y3, Y1) \leftarrow t3(X, Y2, Y1, Y),\ plus(Y2, Y, Y3)$
14. $t3(0, 1, 1, 1) \leftarrow$
15f. $t3(X{+}1, Y2, Y1, Y) \leftarrow t3(X, Y1, Y, Y4),\ plus(Y1, Y4, Y2)$

This final program is linear recursive and it has linear time complexity assuming that the calls of *plus* can be evaluated in constant time. □

Now we briefly describe how the tupling strategy works as indicated in [PrP95]. The eureka definitions needed for deriving the final program are generated by constructing a tree *DefsTree* of definitions, starting from the top level clause of the given program.

In order to indicate how the tree *DefsTree* is constructed, we need the following concepts. (1) Given an expression $R[S]$, a *linking variable* of the subexpression S in R is a variable which occurs both in S and in $R[\_]$. A *local variable* of S in R is a variable which occurs in S and not in $R[\_]$. (2) Given a goal G, we can partition it into the subgoals $G_1, \ldots, G_k$, also called *blocks*, such that: (2.1) any two distinct blocks do not have any variable in common, and (2.2) in each block every two atoms A and B are linked by a list $A, L_1, \ldots, L_n, B$ of atoms of the block such that $n \geq 0$ and any two adjacent atoms on the list have a variable in common. (3) Given a goal G, its *relevant part* w.r.t. a given set T of predicates is the maximal subgoal R of G such that: (3.1) R includes the set G_T of atoms of G whose predicates are in T, and (3.2) R does not include any atom A of $G - G_T$ such that $\exists X_1 \ldots X_n.A$, where $\{X_1, \ldots, X_n\} = vars(A) - vars(G_T)$. (Condition 3.2 tells us that any satisfiable atom, such as A, can be discarded from the body of new definitions because, as indicated in the tupling strategy below, new definitions are constructed from the relevant parts of the goals.) (4) Given a clause C, the *r-blocks of* C are the blocks in which we partition the relevant part of the body of C.

Strategy 1 (Tupling). Given (i) a program P with top level clause C, (ii) a set T of predicates to be tupled, and (iii) a set Ls of replacement laws, we construct a tree *DefsTree* of definitions, called *definition tree*, and a new program *TransfP* as follows.

The root of *DefsTree* is clause C; $NewDefs := \{C\}$; $TransfP := P - \{C\}$;

while $NewDefs \neq \emptyset$ **do**
we take a clause D in *NewDefs*; $NewDefs := NewDefs - \{D\}$;

1. by unfolding, from clause D we derive clauses $E_1, \ldots, E_m$;
2. by applying the replacement laws in Ls, from $E_1, \ldots, E_m$ we derive $F_1, \ldots, F_m$;
   $TransfP := TransfP \cup \{F_1, \ldots, F_m\}$;
3. for each clause F in $\{F_1, \ldots, F_m\}$ whose body has at least one occurrence of predicates in T,
   3a. we construct the r-blocks $G_1, \ldots, G_k$ of F,
   3b. for $i = 1, \ldots, k$, if F cannot be folded w.r.t. G_i using a clause in *DefsTree*, then we consider the new definition $t_i(X_1, \ldots, X_h) \leftarrow G_i$, where t_i is a new predicate symbol and $X_1, \ldots, X_h$ are the linking variables of G_i in F, and
   3c. for each new definition, say N, considered at Point 3b, we expand *DefsTree* by making N a child of D and we add N to *NewDefs*.

endwhile

We update the program *TransfP* as follows: (i) we fold the clauses in *TransfP* w.r.t. the r-blocks which are instances of bodies of clauses in *DefsTree*, and (ii) we remove all clauses on which the top level predicate does not depend.

During the application of the tupling strategy we may also use the equality introduction and elimination rules, in particular when applications of these rules allow subsequent goal replacement or folding steps.

We leave it to the reader to verify that the Recurrence Relation program of Example 2 has been derived by performing the transformation steps indicated in our tupling strategy.

We may provide classes of programs (see, for instance, [PrP95]), where the construction of the definition tree terminates, and thus the tupling strategy is successful. In general, however, 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 in the Recurrence Relation example above. There are cases in which the tupling strategy is not successful because it is not able to construct a *finite* definition tree. In these cases the tupling strategy cannot be used for improving program efficiency.

2.3. A Limitation of the Tupling Strategy

Now we give an example where the tupling strategy is *not* successful. This is due to the fact that the construction of the definition tree does not terminate because every new definition to be introduced in the tree has an increasing number of atoms in the body. Thus, in order to get a linear recursive program we should tuple together a *variable* number of predicate calls. We will see in the following sections that this example can be worked out by using the list introduction strategy.

Example 3 (Binomial Coefficients). Let us consider the following program for computing the binomial coefficients:

1. $t1(I, J, K) \leftarrow b(I, J, K)$
2. $b(I, 0, 1) \leftarrow I \geq 0$
3. $b(I, J, 1) \leftarrow I \geq 0, I = J$
4. $b(I{+}1, J{+}1, K) \leftarrow I > J,\ b(I, J, K_1),\ b(I, J{+}1, K_2),\ plus(K_1, K_2, K)$

whose top level clause is clause 1. This initial program has exponential time complexity because of the two calls of the predicate b in the body of clause 4. In order to derive a linear recursive program we may apply the tupling strategy as described in Section 2.2, where: (i) $\{b\}$ is the set of predicates to be tupled, and (ii) the replacement laws are: (1) the functionality of b, that is, $\forall I, J, K_1, K_2.\ (b(I, J, K_1),\ b(I, J, K_2)) \leftrightarrow (b(I, J, K_1),\ K_1 = K_2)$, and (2) the usual laws of $>$ between integers, such as $\forall I, J.\ (I > J,\ I > J{+}1) \leftrightarrow I > J{+}1$. The definition tree one should construct by applying the tupling strategy has an infinite branch of the form: $D_1, D_2, D_3, \ldots, D_n, \ldots$, where:

D_1. $t1(I, J, K) \leftarrow b(I, J, K)$
D_2. $t2(I, J, K_1, K_2) \leftarrow I > J,\ b(I, J, K_1),\ b(I, J{+}1, K_2)$
D_3. $t3(I, J, K_1, K_2, K_3) \leftarrow I > J{+}1,\ b(I, J, K_1),\ b(I, J{+}1, K_2),\ b(I, J{+}2, K_3)$
$\vdots$
D_n. $tn(I, J, K_1, \ldots, K_n) \leftarrow I > J{+}(n{-}2),\ b(I, J, K_1),\ b(I, J{+}1, K_2),\ \ldots,\ b(I, J{+}(n{-}1), K_n)$
$\vdots$

Clause D_2 is derived from clause D_1 as follows. By unfolding clause D_1, among other clauses, we get the following clause:

5. $t1(I{+}1, J{+}1, K) \leftarrow I > J,\ b(I, J, K_1),\ b(I, J{+}1, K_2),\ plus(K_1, K_2, K)$

Clause 5 has one r-block only. It is: $I > J,\ b(I, J, K_1),\ b(I, J{+}1, K_2)$. Thus, according to Point 3b of the tupling strategy, we introduce clause D_2.

Clause D_3 is obtained from clause D_2 as follows. By unfolding and goal replacement, from clause D_2 among other clauses, we derive:

6. $t2(I{+}1, J{+}1, K_1, K_2) \leftarrow I > J{+}1,\ b(I, J, K_3),\ b(I, J{+}1, K_4),\ b(I, J{+}2, K_5),$
$plus(K_3, K_4, K_1),\ plus(K_4, K_5, K_2)$

The relevant part of clause 6 is the body of clause D_3.

Unfortunately, we are not able to construct a finite definition tree because, for any n, the body of the

clause D_n has n different atoms with predicate b. Thus, the tupling strategy does not terminate and we cannot derive a linear recursive program by applying that strategy.

We will show in Section 3 that by suitable generalisations performed according to our *list introduction strategy* one can obtain a linear recursive program in cases where the tupling strategy is not successful. In particular, for the Binomial Coefficients program our list introduction strategy works as follows. First notice that by using the goal replacement and the equality introduction rules, for any n, clause D_n can be transformed into:

$$D'_n.\quad tn(I,J,K_1,\ldots,K_n) \leftarrow b(I,J,K_1),\ I>J,\ J_1=J+1,\ b(I,J_1,K_2),\ I>J_1,\ J_2=J_1+1,\ \ldots,\ b(I,J_{n-1},K_n)$$

In particular, clauses D_2 and D_3 can be transformed into:

$$D'_2.\quad t2(I,J,K_1,K_2) \leftarrow b(I,J,K_1),\ I>J,\ J_1=J+1,\ b(I,J_1,K_2)$$

$$D'_3.\quad t3(I,J,K_1,K_2,K_3) \leftarrow b(I,J,K_1),\ I>J,\ J_1=J+1,\ b(I,J_1,K_2),\ I>J_1,\ J_2=J_1+1,\ b(I,J_2,K_3)$$

We then introduce a new predicate *b_list* defined as follows:

$$L1.\quad b\_list(I,J,[\,],J) \leftarrow$$

$$L2.\quad b\_list(I,J,[K_1|Ks],J_n) \leftarrow b(I,J,K_1), I>J, J_1=J+1,\ b\_list(I,J_1,Ks,J_n)$$

By using the predicate *b_list* we may replace a goal of the form:

$$b(I,J,K_1),\ I>J,\ J_1=J+1,\ \ldots,\ b(I,J_{m-1},K_m),\ I>J_{m-1},\ J_m=J_{m-1}+1$$

for any $m>0$, by the single atom:

$$b\_list(I,J,[K_1,\ldots,K_m],J_m),$$

and clauses D'_2 and D'_3 may be rewritten as follows:

$$E_2.\quad t2(I,J,K_1,K_2) \leftarrow b\_list(I,J,[K_1],J_1), b(I,J_1,K_2)$$

$$E_3.\quad t3(I,J,K_1,K_2,K_3) \leftarrow b\_list(I,J,[K_1,K_2],J_2),\ b(I,J_2,K_3)$$

By matching the bodies of E_2 and E_3 we introduce the following new definition:

$$G_1.\quad genb(I,J,Ks,M,N) \leftarrow b\_list(I,J,Ks,M),\ b(I,M,N)$$

This clause is obtained by *generalising* the list $[K_1]$ in clause E_2 and the list $[K_1,K_2]$ in clause E_3 to the list variable Ks. Generalisation is a well-known strategy for program derivation (see, for instance, [MaW80, Weg76]), and in our framework it allows us to construct the finite definition trees needed for the derivation of new programs. Indeed, we expand a definition tree when a folding step cannot be performed (see Point 3b of the tupling strategy) and by generalisation we introduce definitions which allow foldings more often because terms are replaced by variables.

Now we may continue the construction of the definition tree. We do not present the details of this construction here. We only mention that the full development of the Binomial Coefficients example is performed according to the list introduction strategy given in Section 3. During the application of the list introduction strategy, the following extra definition is introduced:

$$G_2.\quad genb2(I,J,Ks,N,Ls,P,Q) \leftarrow b\_list(I,J,Ks,M),\ b(I,M,N),\ b\_list(I{+}1,M{+}1,Ls,P),\ b(I{+}1,P,Q)$$

and by using the definitions G_1 and G_2 the following program is derived:

$$t1(I,0,1) \leftarrow I=0$$
$$t1(I,J,1) \leftarrow I\geq 0,\ I=J$$
$$t1(I{+}1,J{+}1,K) \leftarrow I>J,\ genb(I,J,[K_1],M,N),\ plus(K_1,N,K).$$
$$genb(I,0,[\,],0,1) \leftarrow I\geq 0$$
$$genb(I,J,[\,],I,1) \leftarrow I\geq 0, I=J$$
$$genb(I{+}1,J{+}1,[\,],J{+}1,N) \leftarrow I>J,\ genb(I,J,[K],M1,N1),\ plus(K,N1,N)$$
$$genb(I,0,[1|Ks],M,N) \leftarrow I>1,\ genb(I,1,Ks,M,N)$$
$$genb(I{+}1,J{+}1,[K|Ks],M,N) \leftarrow genb2(I,J,[L],H,Ks,N,M),\ plus(L,H,K)$$
$$genb2(I,J,Ks,N,[\,],1,I{+}1) \leftarrow I>J,\ genb(I,J,Ks,I,N)$$

$$genb2(I, J, Ks, N, [\,], P, Q) \leftarrow I > J,\ append(Ks, [N], KsN),\ genb(I, J, KsN, Q, M),\ plus(N, M, P)$$
$$genb2(I, J, Ks, N, [L|Ls], P, Q) \leftarrow I > J,\ append(Ks, [N], KsN),\ genb2(I, J, KsN, M, Ls, P, Q),\ plus(N, M, L)$$

together with the clauses for the predicate *append* which denotes list concatenation:

$$append([\,], L, L) \leftarrow$$
$$append([X|Xs], Ys, [X|Zs]) \leftarrow append(Xs, Ys, Zs)$$

The final program existentially terminates for all goals and thus its correctness is a consequence of Theorem 2.1. It runs in time proportional to $n \times k$ for goals of the form $t(n, k, X)$.

3. The List Introduction Strategy

In this section we formally present our list introduction strategy which extends the tupling strategy presented in Section 2.2. In Section 2.3 we have anticipated that our list introduction strategy may be applied to improve program efficiency when during the construction of the definition tree we generate a list of calls of a predicate, say p, such that any two adjacent calls are related by another predicate, say c. That is, the list introduction strategy may be applied whenever we generate a list of calls of the form:

$$L.\ \ p(V_0), c(V_0, V_1),\ \ p(V_1), c(V_1, V_2),\ \ \ldots,\ \ p(V_{n-1}), c(V_{n-1}, V_n)$$

In these cases we are able to derive new, efficient programs which exploit the interactions among the p calls, by defining a new predicate *p_list* which corresponds to a list of p calls, such that any two adjacent calls are in the relation c and the length of the list is not fixed.

In order to derive even more efficient programs, we consider in the following Definition 1 more specific lists of p calls. In these lists every p call has three arguments satisfying the following conditions: (1) the first argument is *shared* among all p calls, (2) the second argument (except for the first call) is a *local* variable of the list of the p calls, and (3) the third argument is *any* variable.

Definition 1 (Goal List). A goal R occurring in the body of a clause C is said to be a *goal list of length* n (*with* $n \geq 0$) *based on the goal* $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$ iff R is of the form:

$$p(X, Y_0, Z_0), c(X, Y_0, Y_1),\ \ p(X, Y_1, Z_1), c(X, Y_1, Y_2),\ \ \ldots,\ \ p(X, Y_{n-1}, Z_{n-1}), c(X, Y_{n-1}, Y_n)$$

where: (i) $X, Y_0, \ldots, Y_n, Z_0, \ldots, Z_{n-1}$ are distinct variables, and (ii) $Y_1, \ldots, Y_{n-1}$ are local variables of R in C.

The notion of a goal list may be extended by allowing: (i) $p(X, Y_0, Z_0)$ and $c(X, Y_0, Y_1)$ to be conjunctions of atoms, instead of single atoms, and (ii) the arguments of p and c to be (possibly empty) tuples of variables, instead of individual variables. For reasons of simplicity, when presenting the list introduction strategy we will stick to the basic notion of a goal list as given in Definition 1. However, in our examples we feel free to use the notion of goal list in the extended sense.

To see an example of a goal list, let us consider clause D'_3 of Example 3:

$$D'_3.\ \ t3(I, J, K_1, K_2, K_3) \leftarrow b(I, J, K_1),\ I > J,\ J_1 = J+1,\ b(I, J_1, K_2),\ I > J_1,\ J_2 = J_1+1,\ b(I, J_2, K_3)$$

We have that the goal:

$$b(I, J, K_1),\ I > J,\ J_1 = J+1,\ \ b(I, J_1, K_2),\ I > J_1,\ J_2 = J_1+1$$

is a goal list of length 2 based on the goal $b(I, J, K_1),\ I > J,\ J_1 = J+1$, obtained (modulo variable renaming) from the goal list R of Definition 1 by replacing $p(X, Y_0, Z_0)$ by the atom $b(X, Y_0, Z_0)$ and $c(X, Y_0, Y_1)$ by the goal $X > Y_0, Y_1 = Y_0+1$.

In the following definition we present the notion of list introduction by which, given a goal of the form $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$, we define a new predicate *p_list* whose third argument is a list.

Definition 2 (List Introduction). The following two clauses:

$$C1.\ \ p\_list(X, Y, [\,], Y) \leftarrow$$
$$C2.\ \ p\_list(X, Y_0, [Z_0|Zs], Y_n) \leftarrow p(X, Y_0, Z_0),\ c(X, Y_0, Y_1),\ p\_list(X, Y_1, Zs, Y_n)$$

are said to define the predicate *p_list* by *list introduction* from the goal $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$.

An example of list introduction from the goal $b(I, J, K_1),\ I > J,\ J_1 = J+1$, is given in Section 2.3, where we have introduced the predicate *b_list* defined by clauses $L1$ and $L2$.

Now we present two properties of the predicate *p_list*. We will use them in the program derivations presented in Section 4.

Property 1. 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 based on $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$, of the form:

$$p(X, Y_0, Z_0),\ c(X, Y_0, Y_1),\ \ p(X, Y_1, Z_1),\ c(X, Y_1, Y_2),\ \ \ldots,\ \ p(X, Y_{n-1}, Z_{n-1}),\ c(X, Y_{n-1}, Y_n)$$

we have that the following replacement law holds in $M(P)$:

$$P1: \quad \forall X, Y_0, Z_0, \ldots, Z_{n-1}, Y_n.(\exists Y_1, \ldots, Y_{n-1}.\ R) \leftrightarrow p\_list(X, Y_0, [Z_0, \ldots, Z_{n-1}], Y_n)$$

This Property 1 allows us to perform some goal replacement steps which are crucial for the application of the list introduction strategy. For instance, by applying the replacement law $P1$, we may replace in the body of clause D'_3 of Example 3 (see Section 2.3) the goal list $b(I, J, K_1),\ I > J,\ J_1 = J+1,\ \ b(I, J_1, K_2),\ I > J_1,\ J_2 = J_1+1$ by the single atom $b\_list(I, J, [K_1, K_2], J_2)$, and by doing so we derive clause E_3.

Property 2. Let P be a program where the predicate *p_list* is defined by clauses $C1$ and $C2$ above. We have that the following replacement law holds in $M(P)$:

$$P2: \quad \forall X, Y_0, L_1, L_2, Y_n.\ (\ \exists Y_m.\ (p\_list(X, Y_0, L_1, Y_m),\ p\_list(X, Y_m, L_2, Y_n)) \leftrightarrow \exists L.\ (append(L_1, L_2, L),\ p\_list(X, Y_0, L, Y_n))\ )$$

where *append* is the familiar predicate which specifies list concatenation.

Notice that by Property 2 the predicate *p_list* defines a homomorphism w.r.t. the list structure of its third argument in the sense of [BiM87].

Definition 3 (Clause Extension). Let R be a goal list of length r based on goal M, and S be a goal list of length s based on the same goal M such that $r < s$. We say that clause $K \leftarrow Body[S]$ is an *extension* of clause $H \leftarrow Body[R]$.

Definition 4 (List Introduction + Generalisation). Let A be the clause $H \leftarrow Body[R]$, where R is a goal list of length r based on the goal $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$. Let *p_list* be the predicate defined by list introduction from $p(X, Y_0, Z_0), c(X, Y_0, Y_1)$ (see Definition 2). By applying the replacement law $P1$ (see Property 1), clause A can be replaced by the clause: $H \leftarrow Body[p\_list(X, Y_0, [Z_0, \ldots, Z_{r-1}], Y_r)]$. We denote by $\gamma(A)$ the following clause:

$$genlist(V_1, \ldots, V_h, L) \leftarrow Body[p\_list(X, Y_0, L, Y_r)]$$

where: (i) *genlist* is a new predicate symbol, (ii) L is a new variable symbol which generalises the list $[Z_0, \ldots, Z_{r-1}]$, and (iii) $\{V_1, \ldots, V_h\} = vars(Body[p\_list(X, Y_0, L, Y_r)]) \cap vars(H, [Z_0, \ldots, Z_{r-1}])$. We say that clause $\gamma(A)$ has been obtained by *list introduction + generalisation* from clause A.

The condition on the variables of the head of $\gamma(A)$ ensures that every clause which can be folded using A can also be folded using $\gamma(A)$ (after a preliminary application of law $P1$). Examples of clause extension and list introduction + generalisation can be found in the derivation of the Binomial Coefficients program of Section 2.3, and indeed, in that derivation clause D'_3 is an extension of clause D'_2 and clause G_1 is $\gamma(D'_2)$.

Definitions 1–4 introduce all the notions we need for presenting our list introduction strategy. In the list introduction strategy, similarly to the tupling strategy, we construct a definition tree by applying the unfolding, goal replacement and definition rules. Notice, however, the following two differences: (1) Before expanding the definition tree and adding a new leaf, say N, we match N against previously generated definitions, and if N is an extension of a clause A higher up in the definition tree then we introduce a *p_list* predicate and we add $\gamma(A)$, instead of N, as a new leaf clause (see Point 3c1 below). (2) Having introduced the predicate *p_list*, we may perform goal replacement steps by applying also the replacement laws $P1$ and $P2$ (see Point 2 below).

Strategy 2 (List Introduction). Given (i) a program P with top level clause C, (ii) a set T of predicates to be tupled, and (iii) a set *Ls* of replacement laws, we construct a tree *DefsTree* of definitions, called *definition tree*, and a new program *TransfP* as follows.

The root of *DefsTree* is clause C; $NewDefs := \{C\}$; $TransfP := P - \{C\}$;

while *NewDefs* $\neq \emptyset$ **do**
we take a clause D in *NewDefs*; *NewDefs* := *NewDefs* $- \{D\}$;

1. by unfolding, from clause D we derive clauses $E_1, \ldots, E_m$;
2. by applying the replacement laws in *Ls* and also the replacement laws $P1$ and $P2$ for the predicates introduced at Point 3c1 below, from $E_1, \ldots, E_m$ we derive $F_1, \ldots, F_m$;
   TransfP := *TransfP* $\cup \{F_1, \ldots, F_m\}$;
3. for each clause F in $\{F_1, \ldots, F_m\}$ whose body has at least one occurrence of predicates in T:

   3a. we construct the r-blocks $G_1, \ldots, G_k$ of F,

   3b. for $i = 1, \ldots, k$, if F cannot be folded w.r.t. G_i using a clause in *DefsTree*, then we consider the new definition $t_i(X_1, \ldots, X_h) \leftarrow G_i$, where t_i is a new predicate symbol and $X_1, \ldots, X_h$ are the linking variables of G_i in F, and

   3c. for each new definition, say N, considered at Point 3b,
   if N is an extension of the clause $A: \ H \leftarrow \mathit{Body}[R]$, occurring in the path of *DefsTree* from C to D, and R is a goal list based on the goal M

      3c1. **then** (*List Introduction + Generalisation*) (i) we add to P the clauses for the predicate *p_list* defined by list introduction from M, (see Definition 2), (ii) we add the predicate *p_list* to the set T, (iii) we expand *DefsTree* by making the clause $\gamma(A)$ (constructed as in Definition 4) a child of D, and (iv) we add $\gamma(A)$ to *NewDefs*

      3c2. **else** (*Tupling*) we expand *DefsTree* by making N a child of D and we add N to *NewDefs*.

endwhile

Finally, we update the program *TransfP* as follows: (i) we fold the clauses in *TransfP* w.r.t. the r-blocks which are instances of bodies of clauses in *DefsTree* (before performing these folding steps we may need to apply the replacement law $P1$, thereby replacing goal lists by the corresponding *p_list* atoms) and (ii) we remove all clauses on which the top level predicate does not depend.

Similarly to the case of the tupling strategy, during the application of the list introduction strategy we may also use the equality introduction and elimination rule. In particular, when checking that clause N is an extension of clause A, we may first introduce or eliminate equalities as needed. Moreover, we may introduce or eliminate equalities for allowing subsequent goal replacement or folding steps (see, for instance, the Binomial Coefficients and World Series Odds examples).

4. Examples of Program Transformation by List Introduction

Now we present two examples of program transformation relative to the N-Queens problem and the World Series Odds problem. They are seemingly unrelated problems, but their efficient solution can be derived as we now show, by straightforward applications of our list introduction strategy.

4.1. N-Queens Problem

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 *Queens* program, which can be viewed as the formal specification of the given problem. This initial program, similar to the one in [StS94, page 253], 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 list $[1, \ldots, n]$.

Initial 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. $safequeen(Q1, [Q2|Qs]) \leftarrow notattack(Q1, Q2),\ safequeen(Q1, Qs)$

In order to place n queens on an $n \times n$ board so that the resulting configuration is safe, we may use this initial *Queens* program and solve the goal: $queens([1, \ldots, n], Qs)$. By clause 1, this goal reduces to the two subgoals $placequeens([1, \ldots, n], Qs),\ safeboard(Qs)$. The first subgoal generates an $n \times n$ board configuration Qs and the second subgoal verifies that in the configuration Qs no two queens lie on the same diagonal.

We assume that the predicate $notattack(Q1, Q2)$ holds iff the queen position $Q1$, that is, the $\langle$row, column$\rangle$ pair $Q1$, 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 Ns is a list of distinct numbers in $[1, \ldots, n]$, Q is the queen position $\langle R, C\rangle$ such that row R is the length of Ns and 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 queens (with $1 \leq k \leq n$) 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 numbers in $[1, \ldots, n]$.

Our initial *Queens* program is a typical application of the *generate-and-test* programming technique and it is inefficient because many unsafe board configurations are generated. In [StS94, page 255] a more efficient *accumulator* program for the N-Queens problem is proposed. In that program 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 list introduction strategy we will mechanically derive a program which is similar to the accumulator program version of [StS94]. Indeed, this strategy will allow us to realise the so called *filter promotion* described in [Bir84, Dar78], 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 [BDK89], 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 filter promotion can be realised by: (i) tupling together all calls of the predicates which occur in the body of a clause and act on a board configuration, that is, calls of the predicates *queens*, *placequeens*, *safeboard*, and *safequeen*, and (ii) moving the calls of the *notattack* predicate to the left of the calls which are tupled together. In a left-to-right mode of evaluation, as in Prolog, the transformed program avoids the inefficient generate-and-test behaviour because the tests for safeness are performed also for incomplete board configurations.

We apply the list introduction strategy as described in Section 3 where: (i) the top level clause is clause 1, (ii) the set T of predicates to be tupled is $\{queens, placequeens, safeboard, safequeen\}$, and (iii) the set Ls of replacement laws is empty. We construct the definition tree *DefsTree* starting from the root clause 1 and we derive the final program *TransfQueens* as we now describe.

Initially, the set *NewDefs* is {clause 1}. The **while-do** of the list introduction strategy is executed as follows.

First Iteration. By unfolding, from clause 1 we get:

8. $queens([\,], [\,]) \leftarrow$
9. $queens(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),$
   $placequeens(Ns1, Qs),\ safequeen(Q, Qs),\ safeboard(Qs)$

Clause 9 has one r-block only, that is, the goal $placequeens(Ns, Qs),\ safequeen(Q, Qs),\ safeboard(Qs)$. Since clause 9 cannot be folded w.r.t. this r-block, we introduce the following new definition:

10. $t1(Ns, Qs, Q) \leftarrow placequeens(Ns, Qs),\ safequeen(Q, Qs),\ safeboard(Qs)$

Clause 10 is made a child of clause 1 in *DefsTree* and *NewDefs* is {clause 10}.

Second Iteration. By unfolding, from clause 10 we get:

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

Clause 12 cannot be folded w.r.t. its r-block, and thus, we introduce the following new definition:

13. $t2(Ns1,Qs,Q,Q1) \leftarrow placequeens(Ns1,Qs),\ safequeen(Q,Qs),\ safequeen(Q1,Qs),\ safeboard(Qs)$

Clause 13 is an extension of clause 10. Indeed, the bodies of clauses 10 and 13 are equal, except that in the body of clause 10 there is an occurrence of the goal $safequeen(Q,Qs)$, which can trivially be viewed as a goal list of length 1 based on the goal $safequeen(Q,Qs)$, whereas in the body of clause 13 there is an occurrence of the goal $safequeen(Q,Qs), safequeen(Q1,Qs)$ which is a goal list of length 2 also based on $safequeen(Q,Qs)$. By list introduction + generalisation we introduce: (i) the predicate *safequeen_list*, defined by the clauses:

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

and (ii) the following clause $\gamma(10)$:

16. $genlist1(Ns,Qs,Ps) \leftarrow placequeens(Ns,Qs), safequeen\_list(Ps,Qs), safeboard(Qs)$

The definition tree is expanded by making clause 16 a child of clause 10 and *NewDefs* is {clause 16}.

Third Iteration. By unfolding, from clause 16 we get:

17. $genlist1([\,],[\,],Ps) \leftarrow safequeen\_list(Ps,[\,])$
18. $genlist1(Ns,[Q1|Qs],[\,]) \leftarrow select(Q1,Ns,Ns1),$
$placequeens(Ns1,Qs),\ safequeen(Q1,Qs), safeboard(Qs)$
19. $genlist1(Ns,[Q1|Qs],[P1|Ps]) \leftarrow select(Q1,Ns,Ns1),\ notattack(P1,Q1),$
$placequeens(Ns1,Qs),\ safequeen(P1,Qs),$
$safequeen\_list(Ps,[Q1|Qs]),\ safeboard([Q1|Qs])$

Clause 18 can be folded using clause 10. On the contrary, clauses 17 and 19 cannot be folded w.r.t. their r-blocks. Thus, we introduce the following new definitions:

20. $t3(Ps) \leftarrow safequeen\_list(Ps,[\,])$
21. $t4(Ns1,Qs,P1,Ps,Q1) \leftarrow placequeens(Ns1,Qs),\ safequeen(P1,Qs),$
$safequeen\_list(Ps,[Q1|Qs]),\ safeboard([Q1|Qs])$

which are made children of clause 16. *NewDefs* is {clause 20, clause 21}.

Fourth Iteration. By unfolding clause 20 we get:

22. $t3([\,]) \leftarrow$
23. $t3([P|Ps]) \leftarrow safequeen\_list(Ps,[\,])$

We do not generate any new definition from clause 23 because it can be folded using clause 20, thereby getting clause 23f (see below).

Fifth Iteration. By unfolding, from clause 21 we get:

24. $t4(Ns1,Qs,P1,[\,],Q1) \leftarrow placequeens(Ns1,Qs),\ safequeen(P1,Qs), safequeen(Q1,Qs),$
$safeboard(Qs)$
25. $t4(Ns1,Qs,P1,[P2|Ps],Q1) \leftarrow notattack(P2,Q1),\ placequeens(Ns1,Qs),$
$safequeen(P1,Qs),\ safequeen(P2,Qs),$
$safequeen\_list(Ps,[Q1|Qs]),\ safeboard([Q1|Qs])$

Clause 24 can be folded w.r.t. its r-block using clause 16 (by first applying the replacement law $P1$), while clause 25 cannot be folded w.r.t. its r-block. Thus, we consider the following new definition:

26. $t5(Ns1,Qs,P1,P2,Ps,Q1) \leftarrow placequeens(Ns1,Qs),$
$safequeen(P1,Qs),\ safequeen(P2,Qs),$
$safequeen\_list(Ps,[Q1|Qs]),\ safeboard([Q1|Qs])$

Since clause 26 is an extension of clause 21, by list introduction + generalisation we introduce the following new definition $\gamma(21)$:

27. $genlist2(Ns1, Qs, Ps1, Ps2, Q1) \leftarrow placequeens(Ns1, Qs),\ safequeen\_list(Ps1, Qs),$
$safequeen\_list(Ps2, [Q1|Qs]),\ safeboard([Q1|Qs])$

Clause 27 is made a child of clause 21. *NewDefs* is {clause 27}.

Sixth Iteration. By unfolding and by applying the replacement laws $P1$ and $P2$, from clause 27 we derive:

28. $genlist2(Ns1, Qs, Ps1, [\,], Q1) \leftarrow placequeens(Ns1, Qs),\ safequeen\_list(Ps1, Qs),$
$safequeen(Q1, Qs),\ safeboard(Qs)$
29. $genlist2(Ns1, Qs, Ps1, [P2|Ps2], Q1) \leftarrow notattack(P2, Q1),\ placequeens(Ns1, Qs),$
$safequeen\_list(Ps1, Qs),\ safequeen(P2, Qs),$
$safequeen\_list(Ps2, [Q1|Qs]),\ safeboard([Q1|Qs])$

By applying the replacement laws $P1$ and $P2$, we get:

30. $genlist2(Ns1, Qs, Ps1, [\,], Q1) \leftarrow placequeens(Ns1, Qs),\ safequeen\_list([Q1|Ps1], Qs),\ safeboard(Qs)$
31. $genlist2(Ns1, Qs, Ps1, [P2|Ps2], Q1) \leftarrow notattack(P2, Q1),$
$placequeens(Ns1, Qs),\ safequeen\_list([P2|Ps1], Qs),$
$safequeen\_list(Ps2, [Q1|Qs]),\ safeboard([Q1|Qs])$

Clauses 30 and 31 can be folded using clauses 16 and 27, respectively, thereby deriving clauses 30f and 31f (see below). Now *NewDefs* is the empty set and the **while-do** loop of the list introduction strategy terminates.

We apply the replacement law $P1$ to clauses 9, 18 and 19. Then, by folding and removing all clauses on which the predicate *queens* does not depend, we derive the following final program *TransfQueens*:

8. $queens([\,], [\,]) \leftarrow$
9f. $queens(Ns, [Q|Qs]) \leftarrow select(Q, Ns, Ns1),\ genlist1(Ns1, Qs, [Q])$
17f. $genlist1([\,], [\,], Ps) \leftarrow t3(Ps)$
18f. $genlist1(Ns, [Q1|Qs], [\,]) \leftarrow select(Q1, Ns, Ns1),\ genlist1(Ns1, Qs, [Q1])$
19f. $genlist1(Ns, [Q1|Qs], [P1|Ps]) \leftarrow select(Q1, Ns, Ns1),\ notattack(P1, Q1),$
$genlist2(Ns1, Qs, [P1], Ps, Q1)$
30f. $genlist2(Ns1, Qs, Ps1, [\,], Q1) \leftarrow genlist1(Ns1, Qs, [Q1|Ps1])$
31f. $genlist2(Ns1, Qs, Ps1, [P2|Ps2], Q1) \leftarrow notattack(P2, Q1),\ genlist2(Ns1, Qs, [P2|Ps1], Ps2, Q1)$
22. $t3([\,]) \leftarrow$
23f. $t3([P|Ps]) \leftarrow t3(Ps)$

This program performs much less backtracking than the initial program and its operational behaviour is similar to the accumulator program given in [StS94, page 255]. By clause 9f, the first queen position Q is selected and predicate *genlist*1 is called with its last argument bound to the list $[Q]$. This last argument of *genlist*1 stores the board configuration generated so far. When a new queen is placed at position $Q1$ and it is not attacked by the last queen placed at position $P1$ (see clause 19f), the predicate *genlist*2 checks whether or not this queen is attacked by previously placed queens whose positions are in the list Ps (see clauses 30f and 31f). In case it is not attacked, the configuration is updated (see clause 30f), otherwise, by backtracking (see the *select* atom in clause 19f), a different queen position is considered. If no position for the new queen is safe, then by backtracking (see the *select* atoms in clauses 9f and 18f), the position of a previously placed queen, if any, is modified.

Notice that we can simplify the final program *TransfQueens* by removing the atom $t3(Ps)$ from the body of clause 17f, because it is true for all lists Ps. After removing $t3(Ps)$, the predicate *queens* does not depend on $t3$, and we can remove also clauses 22 and 23f.

Computer experiments using SICStus Prolog confirm the expected efficiency improvements. For instance, the initial program finds a solutions for 10 queens in 210 seconds, while the final program takes 3.2 seconds.

4.2. World Series Odds Problem

Let us consider the following *World Series Odds* problem taken from [AHU83, 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 of the sequence. We take the atom $p(I, J, K)$ to denote that team A has probability K of becoming the champion when A needs to win I games in the future to become the champion, and team B needs to win J games in the future to become

the champion. The value of K is assumed to be 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 *World_Series_Odds* program:

1. $t1(I, J, K) \leftarrow p(I, J, K)$
2. $p(0, J{+}1, 1) \leftarrow$
3. $p(I{+}1, 0, 0) \leftarrow$
4. $p(I{+}1, J{+}1, K) \leftarrow p(I, J{+}1, K1),\ p(I{+}1, J, K2),\ ave(K1, K2, K)$

where $ave(K1, K2, K)$ holds iff $K = (K1{+}K2)/2$. Clause 2 says that team A has probability 1 of becoming the champion if it needs to win 0 games in the future and team B needs to win more than 0 games in the future. Analogously, clause 3 says that team A has probability 0 of becoming the champion if it needs to win more than 0 games in the future and team B needs to win 0 games in the future. Clause 4 says that the probability K of team A becoming the champion when A needs to win $I{+}1$ games in the future and team B needs to win $J{+}1$ games in the future can be recursively computed as follows. Let us consider the case where A wins the next game: in this case $p(I, J{+}1, K1)$ holds for some probability $K1$. In the opposite case where A loses the next game, we have that $p(I{+}1, J, K2)$ holds for some probability $K2$. Since the probability that A wins (or loses) the next game is $1/2$, we have that K is $(K1{+}K2)/2$.

The *World_Series_Odds* program requires $\binom{m+n}{n}$ calls of p for evaluating any goal of the form $t1(m, n, K)$, where m and n are positive integers, and K is a variable. This exponential behaviour is due to the fact that the two p calls in the body of clause 4 share common subcalls which are evaluated more than once. Our initial program is a typical example of functional relation which can be computed by applying the dynamic programming technique. By this technique the results of intermediate p calls are stored in a table which is consulted when a new p call has to be evaluated. Thus, common subcalls are evaluated only once.

Our program transformation strategy will automatically derive a solution similar to the one produced by dynamic programming, by introducing a list which holds the results of the p calls evaluated up to a certain point of the computation. This list is introduced by our list introduction strategy.

We apply this strategy starting from: (i) the given *World_Series_Odds* program with top level clause 1, (ii) the set $\{p\}$ of predicates to be tupled, and (iii) the replacement law stating the functionality of p, that is, $\forall I, J, K1, K2.\ (p(I, J, K1), p(I, J, K2)) \leftrightarrow (p(I, J, K1), K1 = K2)$. We construct a definition tree *DefsTree* starting from the root clause 1 and we derive a new program *TransfWSO* as we now describe.

Initially, the set *NewDefs* is {clause 1}. The **while-do** of the list introduction strategy is executed as follows.

First Iteration. By unfolding clause 1 we derive the following clauses:

5. $t1(0, J{+}1, 1) \leftarrow$
6. $t1(I{+}1, 0, 0) \leftarrow$
7. $t1(I{+}1, J{+}1, K) \leftarrow p(I, J{+}1, K1),\ p(I{+}1, J, K2),\ ave(K1, K2, K)$

Clause 7 has one r-block only, that is, the goal $p(I, J{+}1, K1),\ p(I{+}1, J, K2)$. Since clause 7 cannot be folded w.r.t. this goal, we introduce the following new definition:

8. $t2(I, J, K1, K2) \leftarrow p(I, J{+}1, K1),\ p(I{+}1, J, K2)$

which is made a child of clause 1 in *DefsTree*. *NewDefs* is {clause 8}.

Second Iteration. By unfolding clause 8 and then applying the functionality law of p, we derive the following clauses:

9. $t2(0, J, 1, K2) \leftarrow p(1, J, K2)$
10. $t2(I{+}1, 0, K1, 0) \leftarrow p(I, 1, K3),\ p(I{+}1, 0, K4),\ ave(K3, K4, K1)$
11. $t2(I{+}1, J{+}1, K1, K2) \leftarrow p(I, J{+}2, K3),\ p(I{+}1, J{+}1, K4),\ p(I{+}2, J, K5),$
$ave(K3, K4, K1),\ ave(K4, K5, K2)$

Clauses 9 and 10 can be folded using clauses 1 and 8, respectively. On the contrary, clause 11 cannot be folded w.r.t. its r-block, and we consider the following new definition:

12. $t3(I, J, K3, K4, K5) \leftarrow p(I, J{+}2, K3),\ p(I{+}1, J{+}1, K4),\ p(I{+}2, J, K5)$

We notice that, by applying the equality introduction rule, clauses 8 and 12 can be replaced by the following clauses 8e and 12e, respectively:

8e. $t2(I, J, K1, K2) \leftarrow p(I, J1, K1), I1 = I{+}1, J1 = J{+}1,$
$p(I1, J, K2)$

12e. $t3(I,J,K3,K4,K5) \leftarrow p(I,J2,K3), I1{=}I{+}1, J2{=}J{+}2,$
$p(I1,J1,K4), I2{=}I1{+}1, J1{=}J{+}1,$
$p(I2,J,K5)$

and clause 12e is an extension of clause 8e. Thus, by list introduction + generalisation we introduce the following new definition (which is γ(8e)):

13. $genlist1(I,J,Ks,M,N) \leftarrow p\_list(I,J,Ks,L,M),\ p(L,M,N)$

where the predicate *p_list* is defined as follows:

14. $p\_list(I,J,[\,],I,J) \leftarrow$
15. $p\_list(I,J,[K|Ks],L,M) \leftarrow p(I,J,K), I1{=}I{+}1, J{=}J1{+}1,\ p\_list(I1,J1,Ks,L,M)$

Clause 13 is made a child of clause 8 in *DefsTree*. *NewDefs* is {clause 13}.

Third Iteration. By unfolding clause 13 we get the following clauses:

16. $genlist1(I,J,[\,],J,N) \leftarrow p(I,J,N)$
17. $genlist1(0,J{+}1,[1|Ks],M,N) \leftarrow p\_list(1,J,Ks,L,M),\ p(L,M,N)$
18. $genlist1(I{+}1,J{+}1,[K|Ks],M,N) \leftarrow p(I,J{+}1,K1),\ p(I{+}1,J,K2),$
$p\_list(I{+}2,J,Ks,L,M),\ p(L,M,N),\ ave(K1,K2,K)$

Clauses 16 and 17 can be folded using clauses 1 and 13, respectively, thereby deriving clauses 16f and 17f (see below). Clause 18 cannot be folded w.r.t. its r-block and we introduce the new definition:

19. $t4(I,J,K1,K2,Ks,M,N) \leftarrow p(I,J{+}1,K1),\ p(I{+}1,J,K2),$
$p\_list(I{+}2,J,Ks,L,M),\ p(L,M,N)$

which is made a child of clause 13 in *DefsTree*. *NewDefs* is {clause 19}.

Fourth Iteration. By unfolding clause 19 and applying the functionality of p we get the following clauses:

20. $t4(I,0,K1,K2,[\,],0,0) \leftarrow p(I,1,K1),\ p(I{+}1,0,K2)$
21. $t4(I,J{+}1,K1,K2,[\,],J{+}1,N) \leftarrow p(I,J{+}2,K1),\ p(I{+}1,J{+}1,K2),\ p(I{+}2,J,K3),$
$ave(K2,K3,N)$
22. $t4(I,J{+}1,K1,K2,[K|Ks],M,N) \leftarrow p(I,J{+}2,K1),\ p(I{+}1,J{+}1,K2),\ p(I{+}2,J,K3),$
$p\_list(I{+}3,J,Ks,L,M),\ p(L,M,N),\ ave(K2,K3,K)$

Since clause 22 cannot be folded w.r.t. its r-block, we consider the following new definition:

23. $t5(I,J,K1,K2,Ks,M,N) \leftarrow p(I,J{+}2,K1),\ p(I{+}1,J{+}1,K2),\ p(I{+}2,J,K3),$
$p\_list(I{+}3,J,Ks,L,M),\ p(L,M,N)$

By applying the equality introduction rule clauses 19 and 23 can be replaced by the following clauses 19e and 23e, respectively:

19e. $t4(I,J,K1,K2,Ks,M,N) \leftarrow p(I,J1,K1), I1{=}I{+}1, J1{=}J{+}1,\ p(I1,J,K2),$
$p\_list(I1{+}1,J,Ks,L,M),\ p(L,M,N)$
23e. $t5(I,J,K1,K2,Ks,M,N) \leftarrow p(I,J1,K1), I1{=}I{+}1, J1{=}J2{+}1,$
$p(I1,J2,K2), I2{=}I1{+}1, J2{=}J{+}1,\ p(I2,J,K3),$
$p\_list(I2{+}1,J,Ks,L,M),\ p(L,M,N)$

We have that clause 23e is an extension of clause 19e, and by list introduction + generalisation we introduce the following new definition (which is γ(19e)):

24. $genlist2(I,J1,Hs,J,K2,Ks,M,N) \leftarrow p\_list(I,J1,Hs,I2,J),\ p(I2,J,K2),$
$p\_list(I2{+}1,J,Ks,L,M),\ p(L,M,N)$

which is made a child of clause 19 in *DefsTree*. *NewDefs* is {clause 24}.

Fifth Iteration. By unfolding clause 24, applying the functionality of p, and applying the equality introduction rule, we get the following clauses:

25. $genlist2(I,J1,Hs,0,K2,[\,],0,0) \leftarrow p\_list(I,J1,Hs,I2,0),\ p(I2,0,K2)$
26. $genlist2(I,J1,Hs,M{+}1,K2,[\,],M{+}1,N) \leftarrow p\_list(I,J1,Hs,I2,J),$
$p(I2,J,K2),\ L{=}I2{+}1,\ J{=}M{+}1,$
$p(L,M,K3),\ ave(K2,K3,N)$

27. $genlist2(I, J1, Hs, J2{+}1, K2, [K|Ks], M, N) \leftarrow p\_list(I, J1, Hs, I2, J),$
$p(I2, J, K2),\ I3{=}I2{+}1,\ J{=}J2{+}1,$
$p(I3, J2, K3),\ p\_list(I3{+}1, J2, Ks, L, M),$
$p(L, M, N),\ ave(K2, K3, K)$

By applying the replacement laws $P1$ and $P2$ clauses 26 and 27 are replaced by the following two clauses 26r and 27r, respectively:

26r. $genlist2(I, J1, Hs, M{+}1, K2, [\,], M{+}1, N) \leftarrow append(Hs, [K2], Zs),\ p\_list(I, J1, Zs, L, M),$
$p(L, M, K3),\ ave(K2, K3, N)$

27r. $genlist2(I, J1, Hs, J2{+}1, K2, [K|Ks], M, N) \leftarrow append(Hs, [K2], Zs),\ p\_list(I, J1, Zs, I3, J2),$
$p(I3, J2, K3),\ p\_list(I3{+}1, J2, Ks, L, M),$
$p(L, M, N),\ ave(K2, K3, K)$

Clauses 25, 26r, and 27r can all be folded w.r.t. their r-blocks using clauses in *DefsTree*, thereby deriving clauses 25f, 26f and 27f (see below). Thus, *NewDefs* is the empty set and the **while-do** loop of the list introduction strategy terminates.

We now apply the equality introduction rule and the replacement law $P1$ to clauses 7 and 18. Then, by folding and removing the clauses on which predicate $t1$ does not depend we derive the following final *TransfWSO* program:

5. $t1(0, J{+}1, 1) \leftarrow$
6. $t1(I{+}1, 0, 0) \leftarrow$
7f. $t1(I{+}1, J{+}1, K) \leftarrow genlist1(I, J{+}1, [K1], J, K2),\ ave(K1, K2, K)$
16f. $genlist1(I, J, [\,], J, N) \leftarrow t1(I, J, N)$
17f. $genlist1(0, J{+}1, [1|Ks], M, N) \leftarrow genlist1(1, J, Ks, M, N)$
18f. $genlist1(I{+}1, J{+}1, [K|Ks], M, N) \leftarrow genlist2(I, J{+}1, [K1], J, K2, Ks, M, N),\ ave(K1, K2, K)$
25f. $genlist2(I, J1, Hs, 0, K2, [\,], 0, 0) \leftarrow genlist1(I, J1, Hs, 0, K2)$
26f. $genlist2(I, J1, Hs, M{+}1, K2, [\,], M{+}1, N) \leftarrow append(Hs, [K2], Zs),\ genlist1(I, J1, Zs, M, K3),$
$ave(K2, K3, N)$
27f. $genlist2(I, J1, Hs, J2{+}1, K2, [K|Ks], M, N) \leftarrow append(Hs, [K2], Zs),$
$genlist2(I, J1, Zs, J2, K3, Ks, M, N),$
$ave(K2, K3, K)$

This program is linear recursive and it requires $O(m \times n)$ calls of *ave* for evaluating any goal of the form $t1(m, n, K)$. Further improvements can be made to this final program and indeed, the *append* predicate can be removed in favour of cheaper *cons* operations by applying standard transformation techniques [ZhG88].

5. Related Work and Final Discussion

The tupling strategy is a well-established technique for program derivation which uses the 'rules + strategies' approach advocated by Burstall and Darlington in [BuD77]. In the case of logic programming this strategy works by combining together several predicate calls so that their interaction can be exploited and their collective evaluation can be performed more efficiently than the evaluation of the single calls in isolation.

In this paper we have first shown that the tupling strategy has a crucial limitation in that the number of predicate calls which is combined together is fixed, independently of the input. Basically, the tupling strategy corresponds to the introduction of *arrays* of fixed length. Then we have proposed an extension of this strategy which allows us to combine a number of predicate calls which is not fixed and depends on the input. This extension, called *list introduction strategy*, is done by introducing *lists* which represent variable-sized conjunctions of predicate calls. We have presented some examples that illustrate the power of our new, extended strategy. In these examples, by introducing lists we were able to derive efficient programs where non-determinism is reduced (see the N-Queens example) and redundant predicate calls are avoided (see the Binomial Coefficients and World Series Odds examples). The reduction of non-determinism is achieved by realising a form of coroutining among the predicate calls stored in the lists we have introduced, so that backtracking may occur as soon as one of the calls fails, and the avoidance of redundant predicate calls is achieved by eliminating multiple occurrences of equivalent goals.

Now we will mention a few transformation techniques which are related to our list introduction strategy and we will compare those techniques to our strategy.

The idea of enabling coroutining among several predicate calls by encoding conjunctions of goals into lists is also used by the *compiling control* technique proposed in [BDK89]. Compiling control works by first generating the symbolic computation of a given set of predicate calls and then synthesising a new program from that symbolic computation. Thus, compiling control does not follow the 'rules + strategies' approach, and it requires *ad hoc* correctness proofs which we do not need here, because we rely on the correctness of the transformation rules. Moreover, our list introduction strategy allows us to avoid useless intermediate data structures which are present in the conjunction of goals stored in the lists. In this sense, it is also an extension of the strategy for avoiding unnecessary variables presented in [PrP95].

Memoisation [Mic68] is a technique which can be used for avoiding redundant computations by recording intermediate results. In the case of logic programming [War92], redundant predicate calls are avoided by storing in a table already computed answers to goals. This table is then looked up at each new predicate call, before invoking a new evaluation. Our transformational approach can be viewed as a sort of memoisation because we record in lists some of the predicate calls which are needed to evaluate the initial goal. However, contrary to memoisation, during program transformation we manipulate the recorded goals by using the goal replacement rule so that redundant, equivalent goals may be discarded. For instance, in our World Series Odds example we have used the functionality of the predicate p, by which a goal of the form $p(I, J, K1), p(I, J, K2)$ is replaced by the simpler goal $p(I, J, K1), K1 = K2$. Program improvements of this kind cannot easily be realised using the memoisation technique, where the storage and retrieval of answers to solved goals are performed at runtime.

The transformation method by Chin and Hagiya [ChH95] combines tupling, *lambda abstraction* [PeS89], and memoisation. Lambda abstractions are represented as dynamic-sized arrays and memoisation is used to avoid the recomputation of these arrays. This method requires the use of suitable analysis techniques to compute at compile time safe bounds of the sizes of the arrays. The transformation methods based on *finite differencing* [PaK82] and *static incrementalization* [LiS99] make use of invariants to efficiently compute, from a collection of function calls relative to the input x, a new collection of function calls relative to a new input of the form: $x \oplus \delta$, where $\oplus$ is a suitable increment function. Our list introduction strategy uses neither program analysis nor invariant discovery, as it applies purely transformational techniques. However, we need to derive suitable eureka definitions whose form is determined by an analysis of the definition tree generated during the transformation process. The discovery of useful invariants corresponds, in our case, to the derivation of the definitions which allow us to perform folding steps.

Our transformational approach is also related to other approaches considered in the case of functional languages where lists and dynamically extensible structures are introduced. Among them we would like to recall: (i) the techniques of Cohen [Coh83], by which arrays whose dimensions depend on the size of the input are introduced, (ii) the *continuation-based* transformations of Wand [Wan80], whereby one exploits the power of higher-order arguments which store sequences of function calls, (iii) the *accumulation* technique of Bird [Bir84], which works by introducing variables to collect a number of previously computed values. All these techniques follow a *schemata approach*, by which program transformations are expressed as a catalogue of conditional equivalences between program schemata. In order to apply a transformation to a program P, one should match P against a given schema and verify that P satisfies suitable conditions. The schemata approach determines much more concise program derivations than the rules + strategies approach, but the schemata approach has its drawbacks: (i) the matching task is not always easy and in some cases matching may even be undecidable [HuL78], and (ii) the choice of the right schema transformation one should apply often requires a deep insight on the program behaviour and may be difficult to mechanise.

We have not described our list introduction strategy in full detail. Nevertheless, the reader may realise that each individual operation of our strategy is based on a straightforward syntactic analysis of the program transformation process and thus it can easily be mechanised. Indeed, many of these operations have been incorporated into the MAP semi-automatic transformation system [RPP98]. In particular, all transformation rules R1–R5 have already been implemented. It should be noticed, however, that in our transformations we often need to make use of program properties which should be proved in advance before supplying them to the transformation system as replacement laws.

Acknowledgements

A preliminary version of this paper has been presented at the International Working Conference on Algorithmic Languages and Calculi [PeP97]. We would like to thank Danny De Schreye for many fruitful

conversations and for comments on the N-Queens example presented in this paper. We also thank Anna Formica and anonymous referees for reading an earlier draft of this paper.

References

[AHU83] A. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Bir84] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Toplas*, 6(4):487–504, 1984.

[BiM87] R. S. Bird and L. G. L. T. Meertens. 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*, pages 451–457. North-Holland, 1987.

[BDK89] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.

[BuD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[ChH95] W.-N. Chin and M. Hagiya. A transformational method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115, 1995.

[Coh83] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, 1983.

[Dar78] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[Dij82] E. W. Dijkstra. *Selected Writing on Computing: A Personal Perspective*. Springer-Verlag, New York, Heidelberg, Berlin, 1982.

[HuL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[LiS99] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In S. Doaitse Swierstra, editor, *Programming Languages and Systems. 8th European Symposium on Programming, ESOP '99*, Lecture Notes in Computer Science 1576, pages 288–305. Springer-Verlag, 1999.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[Man74] Z. Manna. *Mathematical Theory of Computation*. MacGraw-Hill, 1974.

[MaW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Toplas*, 2:90–121, 1980.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[PaK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[PeP96] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[PeP97] A. Pettorossi and M. Proietti. Program derivation via list introduction. In R. Bird and L.G.L.T. Meertens, editors, *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France, February 17-21, 1997*, pages 296–323. Chapman & Hall, 1997.

[PeP98] A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.

[PeS89] A. Pettorossi and A. Skowron. The lambda abstraction strategy for program derivation. *Fundamenta Informaticae*, XII(4):541–561, 1989.

[PrP95] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[RPP98] S. Renault, A. Pettorossi, and M. Proietti. Design, implementation, and use of the MAP transformation system. R 491, IASI-CNR, Rome, Italy, 1998. `http://www.iasi.rm.cnr.it/~proietti/system.html`

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[StS94] L. S. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1994. Second Edition.

[TaS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[War92] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[Weg76] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering SE*, 2:69–79, 1976.

[ZhG88] J. Zhang and P. W. Grant. An automatic difference-list transformation algorithm for Prolog. In *Proceedings 1988 European Conference on Artificial Intelligence, ECAI '88*, pages 320–325. Pitman, 1988.

Some reflections of Alberto Pettorossi

I studied in Edinburgh in the late 1970s under Rod's supervision and from him I learned the main ideas and techniques of program transformation, a field of computer science which Rod started a few years earlier,

together with John Darlington [BuD77]. Before leaving Rome to go to Scotland, I knew Rod as the inventor of 'structural induction', as mentioned in Manna's book [Man74]. Soon after my arrival at the Department of Artificial Intelligence, I realised that he was going to teach me many important things in the area of programming. They are now an essential part of my professional life.

I am very grateful to Rod for teaching me the golden rules of simplicity and clarity. I still remember some sentences, such as: 'You do not need more than one level of subscripts', 'You should care about the reader: do not make him tired or confused', 'Write a statement that is correct, even if you do not have a formal proof for it'. Sometimes I repeat these sentences to my students and to my friend Maurizio Proietti, co-author of this paper as well as many others.

I received a lot from Rod also from a personal point of view. On many occasions he showed me through concrete examples the importance of being humble, kind, generous and open to people of different culture or background. I regard these examples as special gifts which I treasure as much as the scientific insights he shared with me with great enthusiasm and joy.

This paper comes from Rod's suggestions on how to use generalisation techniques for answering a challenge of Dijkstra [Dij82, pages 215–216]. It comes also from a question Rod asked me after a conversation on the tupling strategy: 'Well, Alberto, now that you have re-invented arrays, why don't you re-invent lists?'

'I will reckon him who taught me this art equally dear to me as my parents' (from the Oath of Hippocrates).