Program Tranformations for non-linear recursive logic programs

Manolis Gergatsoulis Maria Katzouraki

Institute of Informatics and Telecommunications
N.C.S.R 'Demokritos', 153 10 A. Paraskevi Attikis, Greece
e_mail: manolis@estia.nrcps.ariadne-t.gr

Abstract

Fold/unfold program transformation is a powerful technique for improving efficiency of logic programs. A fold/unfold system consists of a few, simple but very powerful transformation rules. Between them the rule for the introduction of new predicate definitions (Eureka), requires the greatest creativity. In this paper we study some possible forms of recursive Eureka definitions and the transformation strategies that result in improvement of the efficiency for a class of nonlinear recursive logic programs. The proposed transformations result in the generalization of the relations defined by the logic procedures.

Key Words : program transformations, optimization, generalization, logic programming

1. Introduction

The needs to make software both efficient and correct are often conflicting. A major technique which aims towards solving this conflict is the *program transformation* technique. Using this technique, programs are initially developed by concentrating only on the understandability, correctness and reliability of them. A set of *correctness preserving transformations* is then employeed to solve the efficiency problem by transforming the initial program into another equivalent and hopefully more efficient one (Debray,1988), (Proietti,1988). Most of the program transformation systems are based on fold/unfold transformations which were developed by Burstall and Darlington (Burstall,1977) in the context of functional languages and were formulated for logic programs by Tamaki and Sato (Tamaki,1984), (Tamaki,1986) in such a way that preserve equivalence of logic programs as defined by the least Herbrand model. Fold/unfold systems are composed from a set of some few but very powerful transformation rules. The main problem related to these systems is that a lot of ingenuity is often needed in choosing the right Eureka definitions as well as to decide a strategy for the application of the transformations since their unrestricted application lead in combinatorial explossion.

In this paper we present (through examples) some possible forms of recursive Eureka definitions as well as the corresponding transformation strategies for a class of logic programs with non-linear recursion. These Eureka definitions generalize the relation defined by the logic programs so as the 'old' procedure is a special case of the generalized one. The recursive new definitions result in the *tupling generalization* of the relation defined by the logic procedure. Tupling generalization is a particular but very interesting case of structural generalization, in which a problem dealing with a term is generalized into a problem dealing with a list of terms. Applying fold/unfold transformations as well as some replacement rules for the relation properties on the new definitions, we may find a new procedure that is linear according to the number of recursive calls in the bodies of its clauses. The new procedure has often better computational behaviour than the initial one and it may be tail recursive or it may be transformed to tail recursive by introducing a new argument which plays the role of an accumulator. *Associativity, distributivity* and *identity elements* of relations have been proved to be very useful in the transformation process.

Section 2 of this paper presents the fold/unfold program transformations. Section 3 presents a form of tupling generalization Eureka definitions. In section 4, tupling generalization is combined with accumulator introduction. In section 5 another form of Eureka definition is introduced. In section 6 we present a variant of the Eureka definitions of section 4. Finally, section 7 presents a summury and conclusion.

2. Fold/Unfold transformations

In this section we present the fold/unfold transformations that we use in this paper, which are based

on (Tamaki,1986). In all sentences of this paper, variables are distinguished from constants by giving them uppercase names.

1) *Eureka Definitions*: Add to the current program a procedure that defines a new predicate in terms of already existing predicates. The new definition may be recursive.

2) *Unfolding*: Let c a clause of the form

$c: \quad A \leftarrow A_1, ..., A_{i-1}, A_i, A_{i+1}, ..., A_k.$

and $c_1, ..., c_m$ are all clauses of a program P, whose heads are unifiable with atom A_i with $_1, ..., _m$ the corresponding most general unifiers. *Unfolding c at A_i* consists in replacing the 'old' clause c by the 'new' clauses $c'_1, ..., c'_m$ such that, if

$c_j: \quad B_j \leftarrow B_{j_1}, ..., B_{j_h}. \quad and \quad B_{jj} = A_{ij}, \quad for\ all\ j \ : \ 1 \leq j \leq m$

then

$c'_j: \quad (A \leftarrow A_1, ..., A_{i-1}, B_{j_1}, ..., B_{j_h}, A_{i+1}, ..., A_k)_j.$

3) *Reversible folding* : Let c a clause of the form

$c: \quad B \leftarrow B_1, ..., B_{h_1}, ..., B_{h_k}, ..., B_n.$

We suppose that there exists a clause c_1 in program P

$c_1: \quad A \leftarrow A_1, ..., A_k$

and a substitution for the variables of A such that $A_i = B_{h_i}$ for $i = 1, k$

Then, *folding $B_{h_1}, ..., B_{h_k}$ of c using c_1* consists in replacing c with c' where

$c': \quad B \leftarrow B_1, ..., (A), ..., B_n$

This transformation can be applied only when c_1 is the only clause in P which can resolve the call (A), giving again the clause c from c'.

4) A *replacement rule* is a rule of the form

$\exists X_1, ..., X_n M_1, ..., M_k \Rightarrow \exists Y_1, ..., Y_m N_1, ..., N_h$

where left and right hand sides have the same set of free variables.

A replacement rule replaces a set of goals in the body of a clause with another set of goals. A replacement rule is said to be *correct* relatively to a program P if for every ground instance of the rule : the conjunction of goals on the left hand side is a logical consequence of the program iff the conjunction of goals on the right hand side is a logical consequence of the program.

Very useful replacement rules are those that refer to the properties of the predicates. Between them we can see the rule for the *associativity* of a relation written as:

$\exists R1\ (qa(X, Y, R1), qa(R1, Z, R)) \ \Rightarrow \ \exists R2\ (qa(Y, Z, R2), qa(X, R2, R))$

The rules for the *right identity element* ι_r and *left identity element* ι_l of a relation :

$qa(X, \iota_r, Y) \Rightarrow X = Y$

$qa(\iota_l, X, Y) \Rightarrow X = Y$

And the rules for the *distributivity* of a relation r over the relation q :

$\exists Z\ (q(X, Y, Z), r(A, Z, R)) \Rightarrow \exists W1\ \exists W2\ (r(A, X, W1), r(A, Y, W2), q(W1, W2, R))$

The application of the above transformation rules preserves (Tamaki,1986) the *meaning* of programs.

3. Tupling generalization

The following procedure $flattree$ collects the values of nodes of a label binary tree into a list.

(1) $flattree(void, [\,])$.

(2) $flattree(t(E, LT, RT), [E|L]) \leftarrow flattree(LT, LL), flattree(RT, RL), append(LL, RL, L)$.

(3) $append([\,], X, X)$.

(4) $append([X|Xs], Y, [X|Z]) \leftarrow append(Xs, Y, Z)$.

Our purpose is to transform this program into an equivalent linear recursive program. For this, we introduce the following recursive Eureka definition.

$(Eur1)$ $f\_tpl([\,], [\,])$.

$(Eur2)$ $f\_tpl([T|Ts], L) \leftarrow flattree(T, FT), f\_tpl(Ts, LTs), append(FT, LTs, L)$.

The completion (Deville 1990) of this definition is

(C) $f\_tpl(TL, L) \leftrightarrow (TL = [\,], L = [\,])\ or$
$(TL = [T|Ts], flattree(T, FT), f\_tpl(Ts, LTs), append(FT, LTs, L))$

Instantiation of TL in (C) with $[T]$ yields

(5) $f\_tpl([T], L) \leftrightarrow false\ or([T] = [T], flattree(T, FT), f\_tpl([\,], LTs), append(FT, LTs, L))$

which simplifies to

(6) $f\_tpl([T], L) \leftrightarrow flattree(T, FT), append(FT, [\,], L)$

Eliminating $append$ ($[\,]$ is a right identity element) and taking the only if part we get

(7) $flattree(T, L) \leftarrow f\_tpl([T], L)$.

Now we try to eliminate $flattree$ form {Eur1,Eur2}. For this we unfold (Eur2) at $flattree$:

(8) $f\_tpl([void|Ts], L) \leftarrow f\_tpl(Ts, LTs), append([\,], LTs, L)$.

(9) $f\_tpl([t(E, LT, RT)|Ts], L) \leftarrow flattree(LT, LL), flattree(RT, RL),$
$append(LL, RL, R), f\_tpl(Ts, LTs), append([E|R], LTs, L)$.

Unfolding (8) at $append$ we take

(10) $f\_tpl([void|Ts], L) \leftarrow f\_tpl(Ts, L)$.

Unfolding (9) at $append([E|R], LTs, L)$ we take

(11) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow flattree(LT, LL), flattree(RT, RL),$
$append(LL, RL, R), f\_tpl(Ts, LTs), append(R, LTs, L)$.

Rearraging goals and applying the associativity property of $append$, (11) is transformed to

(12) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow flattree(LT, LL), \underline{flattree(RT, RL)},$
$\underline{f\_tpl(Ts, LTs), append(RL, LTs, W)}, append(LL, W, L)$.

Now, we fold the underlined atoms in (12) using (Eur2). This yields

(13) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow flattree(LT, LL), f\_tpl([RT|Ts], W), append(LL, W, L)$.

One more folding using (Eur2) yields

(14) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow f\_tpl([LT, RT|Ts], L)$.

The final definition of f_tpl, therefore, is :

(15) $flattree(T, L) \leftarrow f\_tpl([T], L)$.

(16) $f\_tpl([\,], [\,])$.

(17) $f\_tpl([void|Ts], L) \leftarrow f\_tpl(Ts, L)$.

(18) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow f\_tpl([LT, RT|Ts], L)$.

The resulting program is linear recursive (tail recursive). A reason for the increase in efficiency comes form the fact that the calls to $append$ have been evaluated at compile time.

Generalizing the above example we can extract the following transformation strategy.

1) Define an auxiliary recursive definition of the form

($Eur1$) $R\_tpl([\,], \iota)$.

($Eur2$) $R\_tpl([X|Y], Z) \leftarrow R(X, W), R\_tpl(Y, Z1), Q(W, Z1, Z)$.

where Q (a relation called in the recursive clause of the initial program) is an associative relation with identity element ι.

2) Using the identity element of Q, find a relation of the form

$R(X, Z) \leftarrow R\_tpl([X], Z)$.

3) Eliminate R from (Eur2). For this, 3.1) unfold (Eur2) at R, 3.2) Apply folding to the resulting clauses using (Eur2) to eliminate R. Use if needed, replacement rules or unfold auxialiary predicates.

4. Combining tupling generalization and accumulator introduction

The following procedure $sumnodes$ evaluates the sum of the leafs of an unlabeled binary tree.

(1) $sumnodes(leaf(N), N)$.

(2) $sumnodes(t(LT, RT), S) \leftarrow sumnodes(LT, SL), sumnodes(RT, SR), S\ is\ SL + SR$.

As before we define

($Eur1$) $sum\_tpl([\,], 0)$.

($Eur2$) $sum\_tpl([T|Ts], S) \leftarrow sumnodes(T, ST), sum\_tpl(Ts, STs), S\ is\ ST + STs$.

Taking the completion, instantiating and simplifying as before we take

(3) $sumnodes(T, S) \leftrightarrow sum\_tpl([T], S)$.

Unfolding (Eur2) at $sumnodes$ using (1) and (2) we take

(4) $sum\_tpl([leaf(N)|Ts], S) \leftarrow sum\_tpl(Ts, STs), S\ is\ N + STs$.

(5) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sumnodes(LT, SL), sumnodes(RT, SR),$
$ST\ is\ SL + SR, sum\_tpl(Ts, STs), S\ is\ ST + STs$.

Applying the associativity rule for $+$, (5) is transformed to

(6) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sumnodes(LT, SL),$
$\underline{sumnodes(RT, SR), sum\_tpl(Ts, STs), W\ is\ SR + STs}, S\ is\ SL + W$.

Folding, the underlined atoms of (6) and then the body of the resulting clause, using Eur2, we take

(7) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sum\_tpl([LT, RT|Ts], S)$.

The final definition which is equivalent to the original program, is therefore

(8) $sumnodes(T, S) \leftrightarrow sum\_tpl([T], S)$.
(9) $sum\_tpl([\,], 0)$.
(10) $sum\_tpl([leaf(N)|Ts], S) \leftarrow sum\_tpl(Ts, STs), S \ is \ N + STs$.
(11) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sum\_tpl([LT, RT|Ts], S)$.

This program cannot execute tail recursively due to the clause (10). However, we can transform it into a tail recursive one, if we introduce an accumulator. For this, we introduce the following Eureka definition.

($Eur3$) $s\_tpl\_acc(L, Acc, S) \leftrightarrow sum\_tpl(L, S1), S \ is \ Acc + S1$.

Instantiating Acc with 0 and L with $[T]$ in (Eur3), simplifying and combining with (8) we take

(12) $sumnodes(T, S) \leftarrow s\_tpl\_acc([T], 0, S)$.

Now we try to find a recursive definition for s_tpl_acc. For this we unfold (Eur3) at sum_tpl using (9), (10) and (11). We take

(13) $s\_tpl\_acc([\,], Acc, S) \leftarrow S \ is \ Acc + 0$.
(14) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow sum\_tpl(Ts, STs), S1 \ is \ N + STs, S \ is \ Acc + S1$.
(15) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl([LT, RT|Ts], S1), S \ is \ Acc + S1$.

(13) simplifies to

(16) $s\_tpl\_acc([\,], S, S)$.

Applying the associativity replacement rule for + and reordering goals, (14) is transformed to

(17) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1 \ is \ Acc + N, sum\_tpl(Ts, STs), S \ is \ Acc1 + STs$.

Now we fold (17) using (Eur3). This yields

(18) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1 \ is \ Acc + N, sum\_tpl\_acc(Ts, Acc1, S)$.

Folding (15) using (Eur3) we take

(19) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl\_acc([LT, RT|Ts], Acc, S)$.

The final tail recursive definition for $sumnodes$ therefore is

(20) $sumnodes(T, S) \leftarrow s\_tpl\_acc([T], 0, S)$.
(21) $s\_tpl\_acc([\,], S, S)$.
(22) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1 \ is \ Acc + N, sum\_tpl\_acc(Ts, Acc1, S)$.
(23) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl\_acc([LT, RT|Ts], Acc, S)$.

5. Another form of tupling generalization definitions

The following $maxtree$ program finds the maximun leaf value of an unlabeled binary tree. Leaf value may be any number.

(1) $maxtree(leaf(N), N)$.
(2) $maxtree(t(LT, RT), M) \leftarrow maxtree(LT, M1), maxtree(RT, M2), maxof(M1, M2, M)$.
(3) $maxof(M, M2, M) \leftarrow M >= M2$.
(4) $maxof(M1, M, M) \leftarrow M > M1$.

The relation $maxof$ is associative but it does not possesses an identity element. For this reason it is not useful to introduce a Eureka definition of the form of section 3. So we try the following definition.

($Eur1$) $max\_tpl(T, [\,], M) \leftarrow maxtree(T, M)$.
($Eur2$) $max\_tpl(T, [H|Hs], M) \leftarrow maxtree(T, M1), max\_tpl(H, Hs, M2), maxof(M1, M2, L)$.

Instantiating properly the completion of this definition we take

(5) $maxtree(T, M) \leftrightarrow max\_tpl(T, [\,], M)$.

Again we eliminate the call to $maxtree$ from {D1,D2}. For this we unfold (Eur1) and (Eur2) at $maxtree$:

(6) $max\_tpl(leaf(N), [\,], N)$.
(7) $max\_tpl(t(LT, RT), [\,], M) \leftarrow maxtree(LT, ML), maxtree(RT, MR), maxof(ML, MR, M)$.
(8) $max\_tpl(leaf(N), [H|Hs], M) \leftarrow max\_tpl(H, Hs, M2), maxof(N, M2, M)$.
(9) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow maxtree(LT, ML), maxtree(RT, MR),$
$maxof(ML, MR, M1), max\_tpl(H, Hs, M2), maxof(M1, M2, M)$.

Folding $maxtree(RT, MR)$ of (7) using (Eur1) we take

(10) $max\_tpl(t(LT, RT), [\,], M) \leftarrow maxtree(LT, ML), max\_tpl(RT, [\,], MR), maxof(ML, MR, M)$.

Now, we fold (10) using (Eur2). This yields

(11) $max\_tpl(t(LT, RT), [\,], M) \leftarrow max\_tpl(LT, [RT], M)$.

Since $maxof$ is associative, (9) can be transformed to

(12) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow maxtree(LT, ML), \underline{maxtree(RT, MR)},$
$\underline{max\_tpl(H, Hs, M2), maxof(MR, M2, L1)}, maxof(ML, L1, M)$.

Folding the underlined atoms of (12), and the body of the resulting clause using (Eur2) we take

(13) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow max\_tpl(LT, [RT, H|Hs], M)$.

The final program which is equivalent to the initial one consists of clauses

(14) $maxtree(T, M) \leftarrow max\_tpl(T, [\,], M)$.
(15) $max\_tpl(leaf(N), [\,], N)$.
(16) $max\_tpl(leaf(N), [H|Hs], M) \leftarrow max\_tpl(H, Hs, M2), maxof(N, M2, M)$.
(17) $max\_tpl(t(LT, RT), [\,], M) \leftarrow max\_tpl(LT, [RT], M)$.
(18) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow max\_tpl(LT, [RT, H|Hs], M)$.

This program is not tail recursive, due to the clause (16), but we can transform it into a tail recursive one using an accumulator introduction strategy similar with the one used in section 4 (not the same due to the lack of identity element for $maxof$).

6. Tupling with more information in list

The procedure $branch\_sum$ that follows, evaluates a list of numbers each one corresponding to the sum of the node values that are in a path from the root of a labeled binary tree to a tree leaf.

(1) $branch\_sum(leaf(N), [N])$.
(2) $branch\_sum(t(W, LT, RT), L) \leftarrow branch\_sum(LT, LL), branch\_sum(RT, RL),$
$append(LL, RL, L1), lsum(W, L1, L)$.
(3) $lsum(\_, [\,], [\,])$.
(4) $lsum(E, [W|Xs], [W1|Ys]) \leftarrow W1\ is\ W + E, lsum(E, Xs, Ys)$.

This program presents a lot of redundancy since some primitive sum operators are done more times than it is needed. We want to transform this program into an equivalent tail recursive one and to eliminate redundant computations. In the following transformation process we will use the rules for the associativity of $append$ and the distributivity of $lsum$ over $append$, as well as the following replacement rule:

(R) $\exists L2(lsum(E1, L1, L2), lsum(E2, L2, L)) \Rightarrow \exists P(P\ is\ E1 * E2, lsum(P, L1, L))$

Now, we introduce the following recursive Eureka definition.

(Eur1) $b\_tpl([\,], [\,])$.
(Eur2) $b\_tpl([(E, T)|Ts], L) \leftarrow branch\_sum(T, L1), lsum(E, L1, S), b\_tpl(Ts, L2), append(S, L2, L)$.

Instantiating the completion with $E = 0$, and $Ts = [\,]$, and simplifying (0 is a left identity element of $lsum$), we take

(5) $branch\_sum(T, S) \leftarrow b\_tpl([(0, T)], S)$.

Unfolding (Eur2) at $branch\_sum$ using (1) and (2) we take

(6) $b\_tpl([(E, leaf(N))|Ts], L) \leftarrow lsum(E, [N], LE1), b\_tpl(Ts, L2), append(LE1, L2, L)$.
(7) $b\_tpl([(E, t(W, LT, RT))|Ts], L) \leftarrow branch\_sum(LT, LL), branch\_sum(RT, RL),$
$append(LL, RL, L2), lsum(W, L2, L1), lsum(E, L1, LE1),$
$b\_tpl(Ts, L2), append(LE1, L2, L)$.

unfolding (twice) (6) at $lsum$ we take

(8) $b\_tpl([(E, leaf(N))|Ts], L) \leftarrow N1\ is\ E + N, b\_tpl(Ts, L2), append([N1], L2, L)$.

evaluating the call $append([N1], L2, L)$ we take

(9) $b\_tpl([(E, leaf(N))|Ts], [N1|L]) \leftarrow N1\ is\ E + N, b\_tpl(Ts, L)$.

Applying the replacement rule (R) to the calls of $lsum$ in (7) and rearranging the goals in body we take

(10) $b\_tpl([(E, t(W, LT, RT))|Ts], L) \leftarrow M\ is\ W + E, branch\_sum(LT, LL), branch\_sum(RT, RL),$
$\underline{append(LL, RL, L2), lsum(M, L2, LE1)}, b\_tpl(Ts, L2), append(LE1, L2, L)$.

Applying the replacement rule for the distributivity on the underlined atoms of (10) we take

(11) $b\_tpl([(E, t(W, LT, RT))|Ts], L) \leftarrow M\ is\ W + E, branch\_sum(LT, LL), branch\_sum(RT, RL),$
$lsum(M, LL, S1), lsum(M, RL, S2), \underline{append(S1, S2, LE1)},$
$b\_tpl(Ts, L2), \underline{append(LE1, L2, L)}$.

Applying the associativity property of $append$, (11) is transformed to

(12) $b\_tpl([(E,t(W,LT,RT))|Ts],L) \leftarrow M \ is \ W+E, branch\_sum(LT,LL), \underline{branch\_sum(RT,RL)}, lsum(M,LL,S1), \underline{lsum(M,RL,S2)}, \underline{b\_tpl(Ts,L2)}, \underline{append(S2,L2,K)}, append(S1,K,L).$

Now we fold the underlined atoms in (12) using (Eur2), and then we fold the body of the resulting clause again with (Eur2). This yields

(13) $b\_tpl([(E,t(W,LT,RT))|Ts],L) \leftarrow M \ is \ W+E, branch\_sum(LT,LL), b\_tpl([(M,RT)|T],K), lsum(M,LL,S1), append(S1,K,L).$

One more folding using (Eur2) yields

(14) $b\_tpl([(E,t(W,LT,RT))|Ts],L) \leftarrow M \ is \ W+E, b\_tpl([(M,LT),(M,RT)|T],L).$

The final definition for *branch_sum* therefore is

(15) $branch\_sum(T,S) \leftarrow b\_tpl([(0,T)],S).$

(16) $b\_tpl([\,],[\,]).$

(17) $b\_tpl([(E,leaf(N))|Ts],[N1|L]) \leftarrow N1 \ is \ E+N, b\_tpl(Ts,L).$

(18) $b\_tpl([(E,t(W,LT,RT))|Ts],L) \leftarrow M \ is \ W+E, b\_tpl([(M,LT),(M,RT)|Ts],L).$

7. Conclusions

In this paper we study some possible forms of recursive Eureka definitions useful for improving the characteristics of an interesting class of logic programs with non-linear recursion. We also propose strategies for the application of fold/unfold transformations. Using these recursive Eureka definitions we can often transform logic programs with non-linear recursion into linear recusrive ones and in most cases into tail recursive. This transformation usually leads in more efficient and/or less space consumming programs. The application transformations that we propose result in *tupling generalization* of the relation defined by a logic procedure. If the resulting program is not a tail recursive one we can transform it further by introducing extra arguments to it that play the role of accumulators (this is another form of generalization). A source of the increase in efficiency comes from the fact that often some calls are evaluated at compile time during the transformation process.

The generalization process is not new in computer science. Generalization has been used in program synthesis from examples in machine learning, in automatic theorem proving, in program synthesis from specifications (Deville,1990), and in program verification, and in program transformations (Wand,1980).

Heuristics for the introduction of Eureka definitions have been studied by other researchers (Proietti, 1988) but the majority of these works deal with introduction of non recursive new definitions.

Predicate properties such as associativity, distributivity and identity elements as well as the corresponding replacement rules have been proved very useful in the transformation process. Associativity and identity elements have been used also by other researchers (Brough,1987).

8. References

Brough, D. R. and Hogger, C. J. (1987), Compiling associativity into logic programs. *J. Logic Programming*, Vol. 4, No. 4, pp. 345-359.

Burstall, R. and Darlington, J. (1977), A transformation system for developing recursive programs. *J.ACM*, Vol. 24, No. 1, pp. 44-67.

Debray, S. K., (1988), Unfold/fold transformations and loop optimization logic programs. In *SIGPLAN' 88 Conference on Programming Language Design and Implementation*, pp. 297-307.

Deville, Y. (1990), *Logic Programming: Systematic Program Development.* Addison-Wesley Publishing Company,Inc.

Proietti, M. and Pettorossi, A. (1988), *Techniques for the automatic improvement of logic programs.* Report R-231, Instituto di Analisi dei Sistemi ed Informatica, Rome.

Tamaki, H. and Sato, T. (1986), *A generalized correctness proof of the unfold/fold logic program transformation.* Technical Report No 86-4, Dept. of Information Science Ibaraki University, Japan.

Tamaki, H. and Sato, T. (1984), Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pp. 127–138.

Wand, M. (1980), Continuation based program transformation strategies. J.ACM Vol. 27, No 1, pp. 164-180.