Explanation Based Program Transformation.

Maurice Bruynooghe Luc De Raedt Danny De Schreye
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3030 Heverlee, Belgium

Abstract

Fold-unfold is a well known program transformation technique. Its major drawback is that folding requires an Eureka step to invent new procedures.
In the context of logic programming, we present a technique where the folding is driven by an example. The transformation is aimed at programs suffering from inefficiencies due to the repetition of identical subcomputations. The execution of an example is analysed to locate repeated subcomputations. Then the structure of the example is used to control a fold-unfold-transformation of the program. The transformation can be automated. The method can be regarded as an extension of explanation based learning.

1 Introduction.

In [Clocksin 88], a technique is presented for translating clausal specifications of numerical methods into efficient programs. The Horn clause program started from does not compute the result but constructs a term which, after evaluation, yields the result (e.g. (0 + 1) + 1 represents the second fibonacci number). Executing the program for a specific input (n) yields a specific term. Clocksin analyses this term to find common subterms and folds the term into a graph structure where each subterm occurs only once. This graph structure can be considered as a program for a hypothetical dataflow computer. Fixing the input is unpractical for fibonacci, but less harmful for more complex cases (e.g. the number of terms in an approximation of a series, the dimension of a matrix equation, the number of points in an n-point discrete Fourier transform).

The regularity of the resulting graph structures is striking. For a human, it is easy to extend them to a larger n, and it is not so hard to come up with a recursive procedure where n is a parameter computing the same value with the same efficiency. The purpose of this paper is to describe a method, suited for automation, deriving such recursive procedures.

Some general techniques are known to address the inefficiencies due to repeated subcomputations e.g. lemma generation [Kowalski 79] and tabulation techniques [Bird 80]. Such techniques give some improvement but do not yield the optimal algorithmic behavior looked for. The fold-unfold transformation technique [Burstall & Darlington 77] could be used, but the fold step requires a Eureka step, the invention of new procedures.

In the area of machine learning, explanation based learning [Mitchel et al. 86] [De Jong & Mooney 86] has been developed to improve the problem solving behaviour of programs. The similarity between explanation based learning and partial evaluation has been pointed out [Van Harmelen & Bundy 89]. However, the point of explanation based learning is that the example is used to *control* the partial evaluation process.

Our method extends this idea, an example will be used to *control* the fold-unfold transformation process. We argue that the method is suited for automation. Also extensions to cases where subcomputations in the original program are not identical but similar seem to be feasible.

In the next section we give some examples; the third section discusses the automation of the method and we finish with a discussion of possible extensions and of related work.

2 Examples.

Our first example is about the well known and simple problem of computing fibonacci numbers. The program:.

```
(C1) fib (0,0).
(C2) fib (1,1).
(C3) fib (N,F) :- N ≥ 2, N1 is N - 1, N2 is N - 2
fib(N1,F1), fib (N2,F2), F is F1 + F2.
```

Executing a query, e.g. fib (5,F5) yields a proof tree with common subgoals, the relevant part is shown in Fig.l. A subtree for the goal fib (3,2) appears twice. An efficient computation should avoid this repetition. This can be achieved by adding 2, the third fibonacci number as extra output argument to the subgoal fib(4,3). This can be realised by a fold-unfold program transformation. By unfolding fib(4,3) we obtain both occurrences of fib(3,2) in the same goal statement and we can use factoring to eliminate the undesired one.

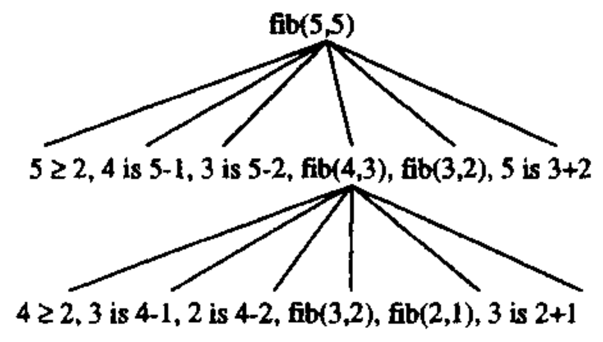
fib(5,5)
5 ≥ 2, 4 is 5-1, 3 is 5-2, fib(4,3), fib(3,2), 5 is 3+2
4 ≥ 2, 3 is 4-1, 2 is 4-2, fib(3,2), fib(2,1), 3 is 2+1

Fig. 1 Part of the prooftree for the query fib(5,F).

So, we take (C3) (the clause used to solve fib (5,F)) and we unfold the call fib(N1,F1) (the call corresponding to fib(4,3)). This yields :

(C31) fib(N,F) :- N $\geq$ 2, N1 is N - 1, N2 is N - 2, {N1 $\geq$ 2, N11 is N1 - 1, N12 is N1 - 2, fib(N11,F11), fib(N12,F12), F1 is F11 + F12 }, fib(N2,F2), F is F1 + F2.

Braces "{", "}" surround the subgoals originating from the unfold step. Comparing (C31) with the example, we have that fib(N11,F11) and fib(N2,F2) correspond to fib(3,2) while fib(N12,F12) corresponds to fib(2,1). So, we apply factoring on fib(N11,F11) and fib(N2,F2): we replace everywhere N2 by N11, F2 by F11 and we remove fib(N2,F2).

(C32) fib(N,F):- N $\geq$ 2, N1 is N - 1, N11 is N - 2, {N1 $\geq$ 2, N11 is N1 - 1, N12 is N1 - 2, fib(N11,F11), fib(N12,F12), F1 is F11 + F12 }, F is F1 + F11.

Now, we fold the subgoals between braces into a new predicate fib1, the arguments are the variables shared with the remainder of (C32), namely N1, F1, N11 and F11. This yields

(C4) fib(N,F):- N $\geq$ 2, N1 is N - 1, N11 is N - 2, fib1(N1,F1,N11,F11), F is F1 + F11.

and, with a bit of renaming:

(C5) fib1(N,F,N1,F1):- N $\geq$ 2, N1 is N - 1, N2 is N -2, fib(N1,F1), fib(N2,F2), F is F1 + F2.

Notice that a more elegant formulation would be obtained by simplifying (C32). Indeed, obviously N11 is N - 2 succeeds iff N1 is N - 1, N11 is N1 - 1 succeed and it could be dropped. As such simplifications in general require a certain reasoning capability, we prefer not to perform them to illustrate that our method does not depend upon such a reasoning capability.

Instead of (C5), we would like to have a recursive clause defining fib1. To obtain such a clause, we observe that we have derived (C5) by unfolding, factoring and folding on the pair fib(N1,F1), fib(N2,F2) in (C3), a pair which can be considered as being obtained by unfolding (using (C3)) a call fib(N,F). Now, (C5) contains an equivalent pair, obtained by unfolding (using (C3)) an equivalent call fib(N1,F1). So, we perform the same unfold and factoring operation on the body of (C5). Unfolding fib (N1,F1) with (C3) yields

(C51) fib1(N,F,N1,F1):- N $\geq$ 2, N1 is N - 1, N2 is N -2, {N1 $\geq$ 2, N11 is N1 - 1, N12 is N1 - 2, fib(N11,F11), fib(N12,F12), F1 is F11 + F12 }, fib(N2,F2), F is F1 + F2.

Factoring fib(N11, F11) and fib(N2,F2) yields

(C52) fib1(N,F,N1,F1):- N $\geq$ 2, N1 is N - 1, N11 is N -2, { N1 $\geq$ 2, N11 is N1 - 1, N12 is N1 - 2, fib(N11, F11), fib(N12,F12), F1 is F11 + F12}, F is F1 + F11.

Because we started from an equivalent pair and performed the same steps, the set of goals between braces in (C52) is necessarily the same as in (C32) (up to renaming). That means we can fold by using (C5) and we obtain the desired recursive clause :

(C6) fib1(N,F,N1,F1):- N $\geq$ 2, N1 is N - 1, N11 is N -2, fib1(N1,F1,N11,F11), F is F1 + F11.

In the realm of explanation based learning, the clauses (C4), (C5) and (C6) are useful clauses which can be added to the program to improve its problem solving behaviour. From a program transformation point of view, the question arises whether the clause (C3) which caused the inefficiencies can be dropped altogether without changing the meaning of fib, or, which other clauses have to be added to allow for the elimination of (C3) (and (C5)).

Given a query fib(N,F) one can originally use the clauses (C1), (C2) and (C3) to solve it. We have manipulated (C3) to obtain new clauses. Obviously, the clauses (C1) and (C2) must be retained.

In manipulating (C3), we have unfolded fib(N1,F1) using (C3) (yielding (C31)). To be able to drop (C3), we have the unfold fib(N1,F1) also with (C1) and (C2).Using (C1), we obtain :

fib(N,F):- N $\geq$ 2, 0 is N - 1, N2 is N - 2, fib(N2,F2), F is 0 + F2.

Since N=1, the test N $\geq$ 2 always fails, so this clause can be dropped.

Using (C2), we obtain :

fib(N,F):- N $\geq$ 2, 1 is N - 1, N2 is N - 2, fib(N2,F2), F is 1 + F2.

This simplifies to :

(C7) fib (2,1).

The clause (C3) can be replaced by the set (C31), (C7). By factoring we derived (C32) from (C31). Can we drop (C31) without losing solutions? This is only the case if, in every proof of fib(N,F) using (C31), it is the case that the instances of fib(N11,F11) and fib(N2,F2) are identical.

Proving this requires the reasoning capability which is typical for program transformation systems. In (C31) we have N1 is N - 1 and N11 is N1 - 1 on the one hand and N2 is N - 2 on the other hand, so obviously N11=N2. Also it is easy to check that fib (N,F) is functional for a given N [Debray & Warren 86], so both instances are always identical and (C32) replaces (C31).

From (C32) to the pair (C4), (C5) is simply a problem reformulation. So, the set of clauses (C1), (C2), (C7), (C4), (C5) is also a complete definition of fib.

To replace (C5) in this set, we follow the same pattern of reasoning, we unfold fib(N1,F1) in (C5) using the complete definition consisting of the set (C1), (C2), (C3). With (C1), we again obtain (after simplification) a test $1 \geq 2$ which always fails.

With (C2), we obtain :

fib1(N,F,1,1):- N ≥ 2, 1 is N - 1, N2 is N - 2, fib(N2,F2), F is 1+F2.

This simplifies to :

(C8) fib1(2,1,1,1).

Thus, (C5) can be replaced by the pair (C51), (C8). Again we can show that the factoring used to derive (C52) from (C51) does not cause loss of solutions while (C6) is a reformulation of (C52).

So, we conclude that the pair (C6), (C8) can replace (C5) and the new program is :

(C1) : fib (0,0).

(C2) : fib (1,1).

(C7) : fib (2,1).

(C4) : fib(N,F):- N ≥ 2, N1 is N - 1, N11 is N - 2, fib1(N1,F1,N11,F11), F is F1 + F11.

(C8) : fib1(2,1,1,1).

(C6) : fib1(N,F,N1,F1):- N ≥ 2, N1 is N - 1, N11 is N -2, fib1(N1,F1,N11,F11), F is F1 + F11.

Finally, using the reasoning capability of a transformational system, we can observe that (C4) and (C6) necessarily fail for N=2, so, a test N ≥ 3 would be more appropriate. Also, using (C6) to fold the body of (C4), one could obtain fib (N,F) :- fib1(N,F,N1,F1).

3 Automation.

In the proof tree of the example, one looks for a pair of calls P,P' for the same predicate such that P' descends immediately from P, the clause used to solve P' is the same as used to solve P and the subtree rooted at P' contains parts also occurring elsewhere in the tree rooted at P (i.e. there is redundancy). (In case of indirect recursion, P' is not an immediate descendant of P, by unfolding one can obtain direct recursive clauses and one can restructure the example prooftree accordingly). The clause used to solve P is of the form :

(C1) P <- α , P' , β where α and β are lists of subgoals.

As dictated by the example, the clause (C1) is unfolded such that the identical calls appear side by side in the unfolded body. This yields

(C11) P <- $\alpha 1$, { γ }, $\beta 1$ where γ is either P' or the list of goals originating from unfolding P'.

Applying the factoring as dictated by the example yields:

(C12) P <- $\alpha 2$, { γ }, $\beta 2$. (γ is left untouched, the redundant calls are removed from $\alpha 1$ and $\beta 1$). This allows to introduce a new predicate P1 and to derive :

(C2) P <- α , P1, $\beta 2$.

(C3) P1 <- { γ }.

In C3, γ originates from unfolding a call P using (C1) so, γ is necessarily a renaming of α P' β or of an unfolding of α P' β. By further unfolding, one can obtain

P1 <- $\alpha 2'$ { γ' } $\beta 2'$

with $\alpha 2'$ renaming of $\alpha 2$, γ' of γ and $\beta 2'$ of $\beta 2$. So, we can use (C3) to fold γ', this yields :

(C4) P1 <- $\alpha 2'$, P1, $\beta 2'$

which is the recursive clause for P1.

To illustrate the automation, we show some more examples. The first one is also taken from [Clocksin 88] and is about the series expansion of the exponential function. The program :

(C1) exp(_,0,1).

(C2) exp(X,N,R):- N ≥ 1, power(X,N,XP), fact(N,NF), N1 is N - 1, exp(X,N1,P1), R is XP/NF + R1.

(C3) power(X,0,1).

(C4) power(X,N,XP):- N ≥ 1, N1 is N - 1, power(X,N1,P1), XP is X * P1.

(C5) fact(0,1).

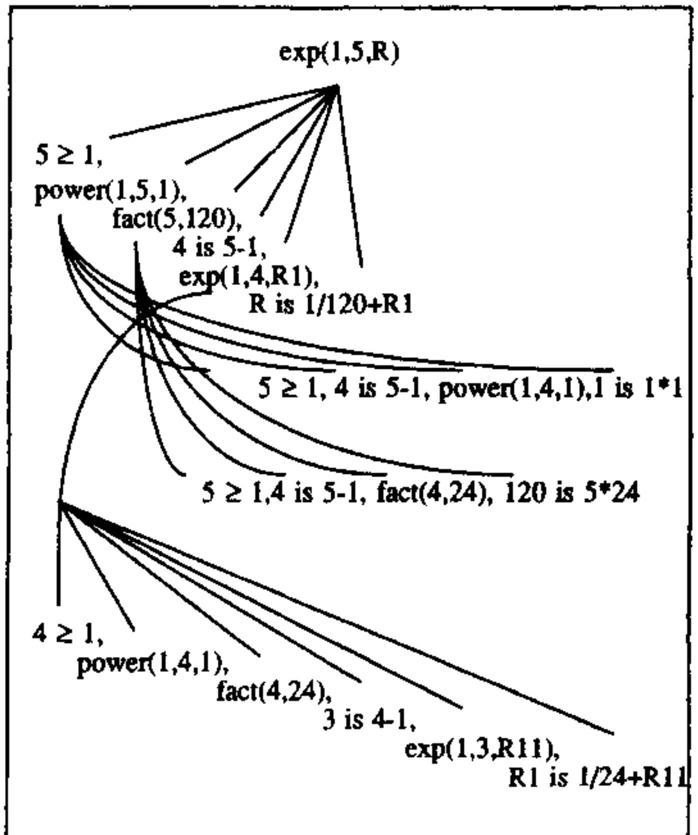

Fig.2 Part of the prooftree for exp(1,5,R).

(C6) fact(N,NF):- N ≥ 1, N1 is N - 1, fact(N1,F1), NF is N * F1.

Part of the proof tree for a query exp(1,5,R) is shown in Fig.2. The shown part exhibits two occurrences of power(1,5,1), two of fact(4,2,4), three of 4 is 5 - 1 and three of 5 ≥ 1. We identify exp(1,5,R) and exp(1,4,R1) as the pair of calls P,P', so we start from C2, the clause used to solve P (and P'). As dictated by the example, we unfold power, fact and exp and obtain:

(C21) exp(X,N,R) :- N ≥ 1, N ≥ 1, N1' is N - 1, power(X,N1',P1), XP is X * P1, N ≥ 1, N1'' is N - 1, fact(N1'',F1), NF is N * F1, N1 is N - 1,{N ≥ 1, power(X,N1,XP1), fact(N1,NF1), N11 is N1 - 1, exp(X,N11,R11), R1 is XP1/NF1 + R11}, R is XP/NF + R1.

As dictated by the example, we apply factoring between fact(N1'', F1) and fact(N1, NF1), between power(X,N1', P1) and power(X,N1,XP1), between N1' is N - 1, N1'' is N - 1 and N1 is N - 1, and between the occurrences of N ≥ 1. We obtain :

(C22) exp(X,N,R):- N ≥ 1, XP is X * XP1, NF is N * NF1, N1 is N - 1,{N1 ≥ 1, power(X,N1,XP1), fact(N1,NF1), N11 is N1 - 1, exp(X,N11,R11), R1 is XP1/NF1 + R11}, R is XP/NF + R11.

We restructure this in the following two clauses :

(C7) exp (X,N,R):- N ≥ 1, XP is X * XP1, NF is N * NF1, N1 is N - 1,exp1(X,N1,R1,XP1,NF1), R is XP/F + R1.

(C8) exp1(X,N,R,XP,NF):- N ≥ 1, power(X,N,XP), fact(N,NF), N1 is N - 1, exp(X,N1,R1), R is XP/NF + R1.

The body of (C8) is the γ part, which is, up to renaming, identical to the body of (C2). Again, we unfold and factor and obtain:

(C82) exp1(X,N,R,XP,NF):- N ≥ 1, XP is X * XP1, NF is N * NF1, N1 is N - 1, {N1 ≥ 1, power(X,N1,XP1), fact(N1,NF1), N11 is N1 - 1, exp(X,N11,R11), R1 is XP1/NF1 + R11}, R is XP/NF + R11.

With (C8), it can be folded into :

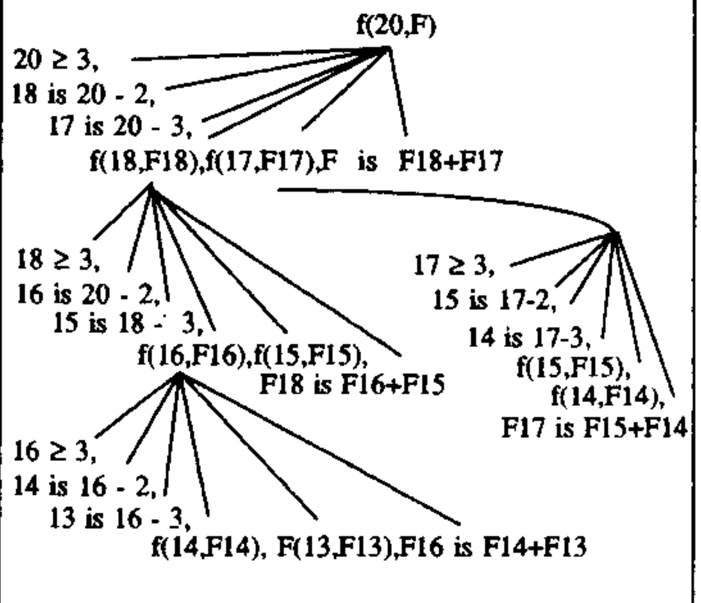

Fig.3 Part of the prooftree for the query f(20,F).

(C9) exp1(X,N,R,XP,NF):- N ≥ 1, XP is X * PP1, NF is N * NF1, N1 is N - 1, exp1(X,N1,R1,Xp1,NF1), R is XP/NF + R11.

Notice, in this case, as well as in the case of fibonacci, the second unfold-factor-fold can easily be avoided, the body of (C9) is the body of (C7), the head is the head of (C8).

We finish this section with a more complex variant on fibonacci :

(C1) f(0,0).
(C2) f(1,1).
(C3) f(2,1).
(C4) f(N,F):- N ≥ 3, N2 is N - 2, N3 is N - 3, f(N2,F2), f(N3,F3), F is F2 + F3.

An example computation is shown in Fig.3. The subgoals f(15,F15) and f(14,F14) of f(17,F17) also occur in the subtree of f(18,F18). So, we designate the calls f(20,F) and f(18,F18) as the pair P, P'. As dictated by the example, we unfold f(N2,F2) and f(N3,F3), while the first call to f created by f(N2,F2) is also unfolded. This yields :

(C41) f(N,F):- N ≥ 3, N2 is N - 2, N3 is N - 3, {N2 ≥ 3, N22 is N2 - 2, N23 is N2 - 3, {N22 ≥ 3, N222 is N22 - 2, N223 is N22 - 3, f(N222,F222), f(N223,F223), F22 is F222 + F223}, f(N23,F23), F2 is F22 + F23 }, N3 ≥ 3, N32 is N3 - 2, N33 is N3 - 3, f(N32,F32), f(N33,F33), F3 is F32 + F33, F is F2 + F3.

Applying factoring between f(N222,F222) and f(N33,F33), between f(N23,F23) and f(N32,F32) yields:

(C42) f(N,F):- N ≥ 3, N2 is N - 2, N3 is N - 3, {N2 ≥ 3, N22 is N2 - 2, N23 is N2 - 3, {N22 ≥ 3, N222 is N22 - 2, N223 is N22 - 3, f(N222, F222), f(N223, F223), F22 is F222 + F223, f(N23,F23), F2 is F22 + F23}, N3 ≥ 3, N23 is N3 - 2, N222 is N3 - 3, F3 is F23 + F222, F is F2 + F3.

Restructuring yields :

(C5) f(N,F):- N ≥ 3, N2 is N - 2, N3 is N - 3, f1(N2,F2,N23,F23,N222,F222), N3 ≥ 3, N23 is N3 - 2, N222 is N3 - 3, F3 is F23 + F222, F is F2 + F3.

(C6) f1(N,F,N3,F3,N22,F22):- N ≥ 3, N2 is N - 2, N3 is N - 3, { N2 ≥ 3, N22 is N2 - 2, N23 is N2 - 3, f(N22,F22), f(N23,F23), F2 is F22 + F23}, f(N3,F3), F is F2 + F3.

Now, the body of (C6) is not the body of (C4), but that body with f(N2,F2) unfolded. Further unfolding of f(N22,F22) and f(N3,F3) yields again a renaming of the body of (C41), we can apply factoring and can use (C6) to fold, this yields :

(C7) f1(N,F,N3,F3,N22,F22):- N ≥ 3, N2 is N - 2, N3 is N - 3, f1(N2,F2,N23,F23,N222,F222), N3 ≥ 3, N23 is N3 - 2, N222 is N3 - 3, F3 is F23 + F222, F is F2 + F3.

which is the recursive clause looked for.

To obtain a completely transformed program, one has to analyse alternative solution paths (using other clauses to unfold) as we did for fibonacci. Due to space constraints, we omit this in this section, as this is in the realm of classic program transformation work.

4 Discussion.

[Clocksin 88] shows a technique to derive an efficient dataflow graph from clausal programs exhibiting redundancy by recomputing identical subgoals several times. The graph is derived for a constant value of one of the inputs. In this paper, we go substantially further, we have shown a technique to obtain an efficient clausal program for the above class of programs. Moreover, the program can be executed for any value of the input which is frozen by Clocksin. The idea underlying the method is to use an example to control an unfold-factoring-fold transformation of the program.

The method as presented requires that subcomputations are identical in the example computation. We are currently investigating whether this condition can be relaxed. A simple example is the towers of hanoi problem. In an example computation one gets subgoals of the form hanoi (5, peg A, peg C, peg B, [...moves]) and hanoi (5, peg C, peg B, peg A, [... moves]), where proofs are structurally identical, only the names of pegs are different. The least general generalisation hanoi (5,X,Y,Z,[...]) still yields the same proof structure, so one can execute that call, take two copies of its successfull instance and unify the first copy with the first call, the second copy with the second call.

Our Explanation Based Program Transformation (EBPT) borrows ideas from Explanation Based Learning (EBL) [Mitchell et al. 86] [De Jong & Mooney 86] [Kedan-Cabelli & Mc Carthy 87] as it is also a form of example guided unfolding. The relationship between partial evaluation and EBL has been studied in [Van Harmelen & Bundy 88]. Our EBPT not only applies example guided unfolding but realises also example guided folding and can introduce new predicates. Consequently, it can modify the structure of the prooftree. Also [Shavlik & De Jong 87 a,b] have developed a method which restructures the prooftree in case of repeated application of the same rule.

It is interesting to observe that restructuring the prooftree is essential to obtain truly operational predicates for the examples we have shown.

We have also borrowed ideas from the area of program transformation. An interesting aspect is the problem of maintaining completeness. Traditional EBL systems derive new rules and add them to the knowledge base but never remove old rules. For certain queries, the new rules may allow to quickly find a first solution, but the amount of redundancy and the total size of the search space increases. Techniques from the area of program transformation allow to prove the equivalence between sets of rules. In EBL this could be used to remove old rules with a bad performance.

The problem we address has also been studied in the area of program transformation. The unfold/fold transformation method of Darlington and Burstall [Burstall & Darlington 77] has served as a general framework for almost every source-level transformation technique proposed for logic- or functional programs. The major advantage of the method is its wide range of applications. These include the introduction of tail-recursion (e.g. [Debray 86], [Pelhat 87]), loopmerging (e.g. [Gregory 80], [Debray 87], [Proietti & Pettorossi 88]), avoiding redundant computations (e.g. [Gregory 80].. [Fronhofer 87]), partial evaluation (e.g. [Komorowski 81], [Venken 84]) and the compilation of control information (e.g. [Gregory 80], [Bruynooghe et al. 86], [De Schreye & Bruynooghe 89]). Closely related to this advantage, is the major drawback of unfold/fold : the method is hard to automate. In general, the degree of automation obtained in any of the applications above is inversely proportional to the size of the class of unfold/fold transformations it can deal with.

Systems designed to support a large class of transformations, either depend on:

- user interaction (e.g. [Gregory 80]),
- a user-provided control program (e.g. [Sato 84]),
- a set of heuristics and global transformation strategies (e.g. [Proietti & Pettorossi 88],

mainly to guide the unfolding, to provide lemma's that can be applied and to generate the new predicates and their definitions (Eureka's) on which the folding can be performed. As such, the techniques described in these three papers can deal with the transformations described in this paper as well, but only under the assumption that the user has supplied the proper directives, control program or heuristics. In contrast with the above, the technique proposed here focuses on a more specific class of transformations. In exchange, we obtain full automation.

[Fronhofer 87] has also studied the problem of eliminating the inefficiencies due to the occurrence of similar subcomputations. The prooftree of an example computation is scanned bottom up for identical subcomputations. At points where the subcomputations start to diverge, an attempt is made to generalise the two observed subcomputations. This generalisation requires an Eureka ; possible automation is not discussed.
A classic approach to avoid redundancy due to identical subcomputations is the use of tabulation (see [Bird 80] for a survey) or lemma generation [Kowalski 79]. The idea is to build a table with answers computed so far. When a new call in encountered, the table is consulted and the stored answer is used when already present. However, it is more a programming technique than a method for program transformation. Also, in logic programming, building the table requires the use of assert which is very time consuming.

Acknowledgements.

M.Bruynooghe is supported by the Belgian National Fund for Scientific Research. D.De Schreye is supported by the Belgian IWONL-IRSIA under contract No. 5203.

References.

[Bird 80] Bird R.S., Tabulation techniques for recursive programs, ACM Computing Surveys, Vol.12, No.4, 1980,pp.:403-417.

[Bruynooghe et al 86] Bruynooghe M., De Schreye D. and Krekels B., Compiling Control, Proc.Third International Symposium on Logic Programming, 1986, pp.

70-78, revised version Journal Logic Programming 1989, pp: 135-162.

.[Bmstall & Darlington 77] Burstall R.M. and Darlington J., A transformation system for developing recursive programs, JACM, 24,1977, pp. 44-67.

[Clocksin 88] Clocksin, W.F., A technique for translating clausal specifications of numerical methods into efficient programs, Journal of Logic Programming, Vol. 5. No. 3,1988, pp231-242.

[Debray & Warren 86] Debray, S.K., Warren, D.S., Detection and optimisation of functional computations in Prolog, Proceedings Third International Logic Programming Conference, Springer Verlag, LNCS, Vol. 225,1986, pp. 490-504.

[Debray 86] Debray S.K., Global optimisation of Logic Programs, Ph.D. dissertation, Stony Brook, 1986.

[Debray 87] Debray S.K., Unfold/fold transformations and loop optimisation of Logic Programs, report of Dept. Computer Science, University of Arizona, 1987.

[De Jong & Mooney 86] DeJong, G., Mooney, R., Explanation-based learning : an alternative view, Machine Learning, Vol. 1, No. 2,1986, pp. 145-176.

[De Schreye & Bruynooghe 89] De Schreye D., Bruynooghe M. On the transformation of logic programs with instantiation based computation rules, J.Symbolic Computation, 1989, pp:125-154..

[Fronhofer 87] Fronhofer B., Double work as a reason for inefficiency of programs, Technical report T.U.M. Munchen, 1987.

[Gregory 80] Gregory S., Towards the compilation of annotated logic programs, Res.Report DOC80/16, June 1980, Imperial College.

[Kedar-Cabelli & McCarthy 87] Kedar-Cabelli, ST., McCarthy, L.T., Explanation based generalization as resolution theorem proving, in: Proceedings of the 4th International Workshop on Machine Learning, Irvine, Morgan Kaufmann, 1987, pp. 383-389.

[Komorowski 81] Komorowski H.J., A specification of an abstract Prolog machine and its applications to partial evaluation, Linkoping Studies in Science and Technology, Dissertation No.69, Linkoping University, 1981.

[Kowalski 79] Kowalski, R.A., Logic for problem solving, North-Holland, 1979.

[Mitchell et al. 86] Mitchell, T.M., Keller, R.M., Kedar-Cabelli, ST., Explanation-based generalization : a unifying view, Machine Learning, Vol. 1, No. 1,1986, pp. 47-80.

[Pelhat 87] Pelhat S., Analysis and control of recursivity in Prolog programs, Technical report CRIL, Universite de Paris-sud, 1987.

[Proietti & Pettorossi 88] Proietti M., Pettorossi A., Some strategies for transforming logic programs, report Istituto di Analisi dei Sistemi ed Informatica, Rome, 1988.

[Sato & Tamaki 84] Sato T., Tamaki H., Transformational logic program synthesis, FGCS '84, Tokyo, 1984.

[Shavlik & De Jong 87a] Shavlik, J., De Jong, G., BAGGER : an EBL system that extends and generalizes explanations. Proceedings of the Sixth National Conference on Artificial Intelligence, 1987, pp. 516-520.

[Shavlik & De Jong 87b] Shavlik, J., De Jong, G., An explanation-based approach to generalizing number. Proceedings of the tenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Milano, 1987, pp. 236-238.

[Van Harmelen & Bundy 89] Van Harmelen, F., Bundy, A., Explanation Based Generalization = Partial Evaluation, to appear in : Artificial Intelligence, 1989.

[Venken 84] Venken R., A Prolog Meta-interpreter for partial evaluation and its applications to source to source transformation and query-optimisation, Proc. of the 6th. ECAI, 1984, pp.:91-100.