Some Thoughts on the Role of Examples in Program Transformation and its Relevance for Explanation-based Learning — Source link

Maurice Bruynooghe, Danny De Schreye

Institutions: Katholieke Universiteit Leuven

Published on: 01 Oct 1989 - Lecture Notes in Computer Science (Springer, Berlin, Heidelberg)

Topics: Explanation-based learning, Program transformation and Partial evaluation

Some Thoughts on the Role of Examples in Program Transformation and its Relevance for Explanation-based Learning

Maurice Bruynooghe[*]
Danny De Schreye[**]

Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3030 Heverlee, Belgium

ABSTRACT

Explanation-based learning is using the same technique as partial evaluation, namely unfolding. However, it brings a new insight: an example can be used to guide the transformation process. In this paper, we further explore this insight and show how examples can be used to guide other kinds of program transformation, guiding not only the unfolding, but also the introduction of new predicates and the folding. On the other hand, we illustrate the more fundamental restructuring which is possible with program transformation and the relevance of completeness results to eliminate computationally inefficient knowledge.

1. Introduction

In the area of machine learning, explanation-based learning [Mitchell et al 86], [DeJong & Mooney 86] has been developed as a method to improve the problem solving behaviour of programs. As pointed out in [Van Harmelen & Bundy 89], the technique used in explanation-based learning is basically partial evaluation [Komorowski 81], [Venken 84] or unfolding [Burstall & Darlington 77]. However, explanation-based learning gives an important new insight : an example can be used to control the partial evaluation process.

To the best of our knowledge, example computations have never been explicitly used to control the transformation of programs, although examples did play a role in the technique we developed under the heading *compiling control* [Bruynooghe et al 86]. Actually, we worked hard to reduce the role of the examples and replaced concrete example computations by abstract ones.

In this paper, we explore the role examples could play in program transformation. We start by recalling a technique in which an example is used to control an unfold-fold transformation which eliminates redundant subcomputations [Bruynooghe et al 89a].

We explore the idea further and show how examples can also be used to realize loop merging and to eliminate intermediate data structures. In all these cases, the eureka required by the unfold-fold transformation [Burstall & Darlington 77] is eliminated and the transformation is controlled by the information extracted from the example computation.

Other works addressing the same class of problems are [Gregory 80], [Debray 88], [Proietti &

*) Supported by the Belgian National Fund for Scientific Research.
**) Supported by the Belgian I.W.O.N.L.-I.R.S.I.A. under contract number 5203.

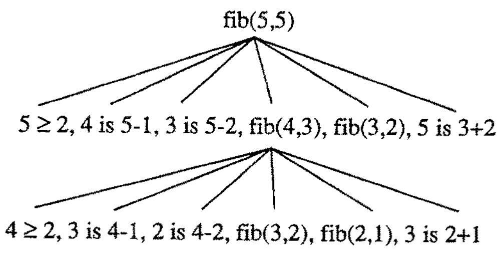
Peterossi 88] for loop merging and [Wadler 84], [Wadler 85], [Wadler 88] and [Debray 88] for the elimination of intermediate data structures.
Attention is also paid to an issue which is usually ignored in the context of example based machine learning : not only is it important to uncover computationally well-behaved clauses, one should also try to eliminate clauses with a computationally poor behaviour. Results from [Tamaki & Sato 84] on equivalence preserving transformations are particularly relevant here.
Finally, we explore the role of examples in our compiling control transformation method.
The paper is written in the framework of logic programming and familiarity with the basic notions is assumed [Kowalski 79].

2. Elimination of redundant subcomputations

A well known example of a program whose top down execution contains several instances of the same subcomputation is the program for computing fibonacci numbers :

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.

Part of an example computation is shown in Fig.1. The prooftree exhibits two instances of the call fib(3,2). The subtrees rooted at these calls (not shown) are identical. Some general techniques i.e. lemma generation [Kowalski 79] and tabulation techniques [Bird 80] can be used to avoid the repetition of the same subcomputation. However, such techniques do not yield the optimal algorithmic behaviour which is possible for such a problem. The classic unfold-fold transformation [Burstall & Darlington 77] requires a eureka step. This eureka can be avoided by making use of the example. The idea is to unfold the recursive clause C3 in such a way that the two identical calls appear in the same body. Then, factoring [Chang & Lee 73], also called function merging [Tamaki & Sato 84] can be used to eliminate one of them.


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).

Consider C3, the clause used to solve fib(5,F) in the example computation. The call fib(N1,F1) corresponds to fib(4,3) while fib(N2,F2) corresponds to fib(3,2). So, we unfold fib(N1,F1), using - as in the example computation - C3.
This yields :

C31: fib(N,F):- N≥2, N1 is N-1, N2 is N-2, {N1≥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).
Notice that C31 contains three calls to fib. The example tells us on which pair to apply factoring. Care is taken not to modify the set of calls between the braces. This set must remain a renaming of the body of C3.

C32: fib(N,F):- N≥2, N1 is N-1, N11 is N-2, {N1≥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 define a new predicate fib1. The body of the defining clause consists of the calls between braces in C31 ; the arguments of the head are the variables which have an occurrence between the braces and in the remainder of C31. With a bit of renaming, intended to simplify the clause, we obtain:

C4: fib1(N,F,N1,F1):- N≥2, N1 is N-1, N2 is N-2, fib(N1,F1), fib(N2,F2), F is F1+F2.

This definition can be used to fold the part between braces in C32. The definition is such that the folding is equivalence preserving [Tamaki & Sato 84]. We obtain:

C5: fib(N,F):- N≥2, N1 is N-1, N11 is N-2, fib1(N1,F1,N11,F11), F is F1+F11.

Notice that a more elegant formulation could have been obtained by first simplifying C32. Indeed, N11 is N-2 is implied by N1 is N-1 and N11 is N1-1, so it could be deleted. This is called goal deletion in [Tamaki & Sato 84]. 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.
C5 defines fib/2 in terms of fib1/4, but this is only half of the work as C4 exhibits the same inefficiency as the original clause C3. It is a feature of the transformation so far that the body of C4 is identical (up to renaming) to the body of C3. So, we perform the same manipulations on C4 as we did on C3.
Unfolding fib(N1,F1) with C3 yields:
C41: fib1(N,F,N1,F1):- N≥2, N1 is N-1, N2 is N-2, {N1≥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:
C42: fib1(N,F,N1,F1):- N≥2, N1 is N-1, N11 is N-2, { N1≥2, N11 is N1-1, N12 is N1-2, fib(N11,F11), fib(N12,F12), F1 is F11+F12}, F is F1+F11.

As we started from the same body, it is no accident that we can fold the part between braces with C4, the definition of fib1. Doing this, we obtain:

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.

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 C4).
Here, we can apply the results of [Tamaki & Sato 84]. When unfolding a call in a clause, the original clause can be dropped when the call is unfolded with all clauses defining the called predicate. In clause C3, we have unfolded fib(N1,F1) with C3. In order to be able to drop C3, we also have to unfold fib(N1,F1) with C1 and C2. Using C1, we obtain:

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

Since N=1, the test N≥2 always fails, so this clause can be dropped. Using C2, we obtain :
fib(N,F):- N≥2, 1 is N-1, N2 is N-2, fib(N2,F2), F is 1+F2.

This simplifies to:
C7: fib(2,1).

So, we can replace the clause C3 by the set {C31,C7}. C32 was derived from C31 by factoring. Can it replace C31? To show that this is equivalence preserving, we have to observe that the two merged calls to fib in C31 have the same value of their first argument. Indeed N1 is N-1 and N11 is N1-1 while N2 is N-2, so N11 and N2 are identical. What remains to be done is showing that fib is functional. Techniques for this are known [Debray & Warren 86]. So, one can apply function merging [Tamaki & Sato 84] which is equivalence preserving.
The definition step adds C4, and as already stated, the folding step is equivalence preserving, so C5 replaces C32. At this point, the program consists of the set {C1,C2,C7,C4,C5}.
To eliminate C4, we also have to unfold fib(N1,F1) with all clauses in the definition. The set {C1,C2,C3} forms a complete definition. We have already unfolded with C3. With C1, we again obtain (after simplification) a test 1 ≥ 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, C4 can be replaced by the pair C41, C8. Again we can show that the factoring which replaces C42 by C41 is equivalence preserving. Finally, C6 replaces C42 by an equivalence preserving folding step.
So, we conclude that the pair C6, C8 can replace C4 and the new program is :
C1: fib (0,0).
C2: fib (1,1).
C7: fib (2,1).
C5: 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 C5 and C6 necessarily fail for N=2, so, a test N≥3 would be more appropriate. A somewhat simpler program could be obtained by using C4, the initial definition for fib1, to fold C3. This is possible because both clauses necessarily have the same body (up to renaming). This yields the following definition for the fib predicate :
C1: fib(0,0).
C2: fib(1,1).
C3: fib(N,F) :- fib1(N,F,N1,F1).

We stress that - once the redundant calls have been detected from the examples prooftree - the technique can be completely automated. For a discussion of this automation and for more examples, the reader is referred to [Bruynooghe et al 89a].

3. Loop merging

The unfold-fold transformation method has also been used to eliminate the overhead of traversing a data structure twice. In the context of functional programming, the distinction is usually made between the case where a data structure is used twice and the case where a so called intermediate data structure is first constructed and subsequently consumed. An example of the first case is a

program traversing a list to make the sum of the elements and traversing the same list for a second time to compute the product of the elements. An example of the latter case is a program to make the sum of the squares of a vector by first constructing a vector of the squares and subsequently making the sum of the elements in the latter vector. P. Wadler [Wadler 84, 85, 88] has proposed techniques to automate the elimination of intermediate data structures in functional programs.
The difference between the two classes almost disappears in the context of logic programming. Indeed the declarative reading does not distinguish between construction and use of the data structure. What counts is the existence of a data structure shared by two procedure calls. Debray [Debray 88] has studied this class and has developed an automated technique called loop fusion.
In this section, we further explore the idea of applying the unfold-fold transformation under guidance of an example and investigate the class of programs where data structures are shared between different procedure calls. Obviously, we do not claim new territory for automation. Our goal is more modest : offering an alternative tool for transforming this class of programs.
A simple, yet practical example for this class of programs is the computation of the scalar product of two vectors.

```
C1:  scalar( Xs,Ys,R):- product(Xs,Ys,Zs), sum(Zs,0,R).
C2:  product([],[],[]).
C3:  product([X| Xs],[Y| Ys],[Z| Zs]):-  Z is X*Y, product(Xs,Ys,Zs).
C4:  sum([],R,R).
C5:  sum([X| Xs],Ac,R):- A is X+Ac, sum(Xs,A,R).
```

The prooftree for an example query is shown in Fig.2. In this prooftree, we do not have multiple occurrences of the same subgoal as in the proof trees of section 2, but multiple occurrences of the same (complex) data structure. The idea underlying the transformation is again to bring the calls containing the same instance of the data structure in the same clause body and then to remove one of the instances.

In fact, the calls containing the same instances are already there in the body of C1. Indeed, Zs is the shared variable. Observe also that the only way to get rid of the multiple instances is to define a new predicate and to use that definition to fold the pair of calls containing only one instance.

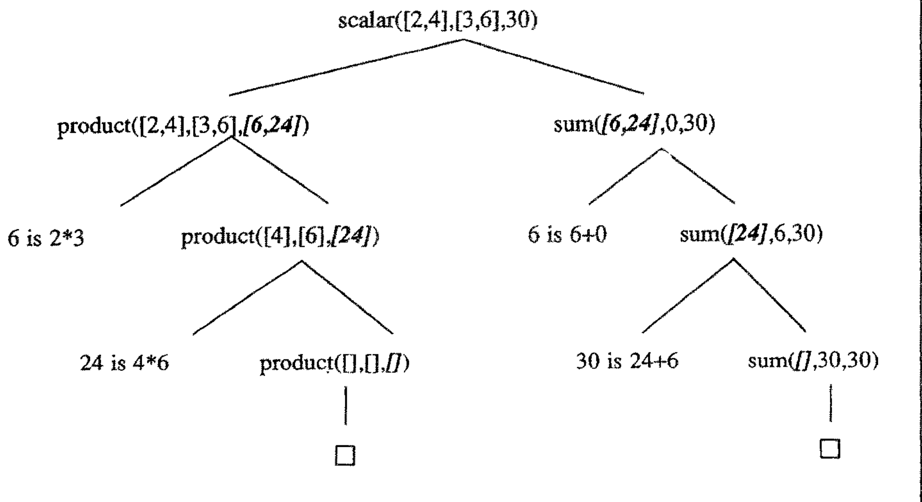


Fig.2: Prooftree for the query scalar([2,4],[3,6],R).

However, the body of C1 is not a good base to define the new predicate as the initialization interferes. So we take C1 and unfold the calls to product and sum using C3 and C5 respectively (as in the example prooftree). This yields:

C6: scalar([X| Xs],[Y| Ys],R):- Z is X*Y, product(Xs,Ys,Zs), A is Z+0, sum(Zs,A,R).

To be able to eliminate one of the occurrences of Zs in C6, we define:

C7: sp(Xs,Ys,A,R):- product(Xs,Ys,Zs), sum(Zs,A,R).

The arguments of the new predicate are the variables shared between the part to be folded and the remainder of C6. Now we can use C7 to fold C6. However, as the body of C1 is an instance of the body of C7, we can also fold C1. This yields

C8: scalar(Xs,Ys,R):- sp(Xs,Ys,0,R).

This folding is equivalence preserving, so C8 replaces C1. The remaining problem is to derive an efficient definition for C7. We apply the same technique and unfold it using C3 and C5.
C9: sp([X| Xs],[Y| Ys],A,R):- Z is X+Y, product(Xs,Ys,Zs), Acc is Z+A, sum(Zs,Acc,R).

Using C7, we can apply an equivalence preserving fold and obtain:

C10: sp([X| Xs],[Y| Ys],A,R):- Z is X+Y, Acc is Z+A, sp(Xs,Ys,Acc,R).

To obtain a complete set of clauses for the sp predicate which can replace C7, we also have to consider other unfoldings. Using C2 and C4 respectively, we obtain:

C11: sp([],[],R,R).

Using the pairs C2, C5 or C3, C4 for the unfolding yields a unification conflict and no other clauses can be derived. So, the new program is :

C8: scalar(Xs,Ys,R) :- sp(Xs,Ys,0,R).
C10: sp([X| Xs],[Y| Ys],R) :- Z is X+Y, Acc is Z+A, sp(Xs,Ys,Acc,R).
C11: sp([],[],R,R).

As another example where the data-structure is traversed twice, but at different speeds, consider the following program.
C1: pairsum(L,S,E):- sum(L,0,S), evensum(L,0,E).
C2: sum([],R,R).
C3: sum([X| L],A,R):- Acc is X+A, sum(L,Acc,R).
C4: evensum([],R,R).
C5: evensum([X],R,R).
C6: evensum([X,Y| L],A,R) :- Acc is Y+A, evensum(L,Acc,R).

An example computation is shown in Fig.3. Again, we have the data structure which is common to different subgoals. The transformation proceeds in a similar way. To eliminate the effects of the initialization, we unfold C1 before defining a new predicate (using C3 and C6 as in the example).
C7: pairsum ([X,Y| L],S,E):- Acc1 is X+0, sum([Y| L],Acc1,S), Acc2 is Y+0, evensum(L,Acc2,E).

To synchronize sum and evensum, another unfolding for sum is needed (as the example prooftree shows):
C8: pairsum([X,Y| L],S,E):- Acc1 is X+0, Acc3 is Y+Acc1, sum(L,Acc3,S), Acc2 is Y+0,
evensum(L,Acc2, E).

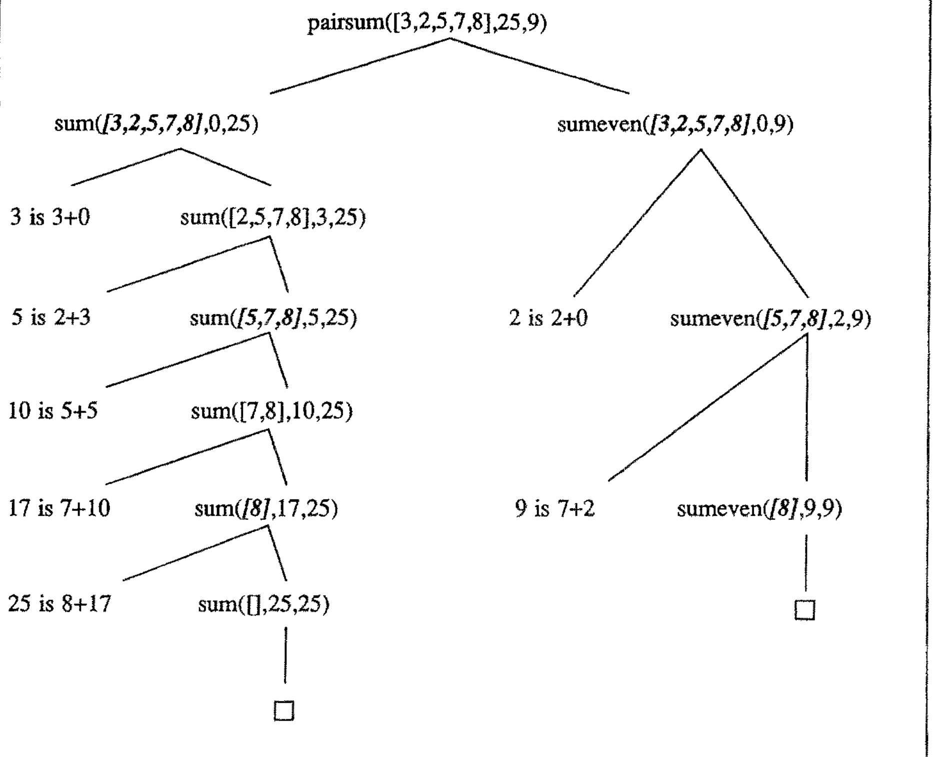


Fig.3: Prooftree for the query pairsum([3,2,5,7,8],S,E).

To eliminate the shared data structure L, we fold sum and evensum into a new predicate:
C9: ps(L,Accs,S,Acce,E):- sum(L,Accs,S), evensum(L,Acce,E).

Notice that L remains as an argument because it appears in the head of C8. Here as well, we can use the new definition C9 to fold the original clause C1:
C10 : pairsum(L,S,E):- ps(L,0,S,0,E).

The transformation is equivalence preserving, so C10 replaces C1. To obtain clauses for the new predicate ps, we have to unfold its definition. Unfolding sum twice and evensum once - as dictated by the example - we obtain a clause which is foldable:
C11: ps([X,Y| L],Accs,S,Acce,E):- Accs1 is X+Accs, Accs2 is Y+Accs1, sum(L,Accs2,S),
Acce1 is Y+Acce, evensum(L,Acce1,E).

The folding with C9 is equivalence preserving and yields:
C12: ps([X,Y| L],Accs,S,Acce,E):- Accs1 is X+Accs, Accs2 is Y+Accs1, Acce1 is Y+Acce,
ps(L,Accs2,S,Acce1,E).

To obtain a complete set of clauses for ps, we also have to consider other unfoldings of C9. Using C2 and C4 yields:
C13: ps([],S,S,E,E).

Using C3, then C2 for sum and C5 for evensum yields :
C14: ps([X| L],Accs,S,E,E):- S is X+Accs.

The above example cannot be handled by [Debray 88] since he unfolds both calls in the initial clause for pairsum only once.
As already noted by Wadler and Debray, not all intermediate datastructures can be eliminated. A procedure like reverse produces its output in one step in the final step of its execution. So, although the output can be shared with another data structure (an example is given in Fig.4), one cannot find sharing of corresponding parts at lower levels of the example prooftree and it is impossible to apply the transformation (unless one observes that the consumer can do its work in a different order).
The distinguishing feature in example computations allowing for the transformation is the occurrence of a complex data structure in two different branches of the prooftree (it need not be identical, it can be a list in one branch and the tail of that list in another branch) and the occurrence of the parts essential for the recursive composition of the data structure at regular points down the branches (in the same relationship as the initial occurrences). Having identified the corresponding point, the example again guides the unfold-fold transformation. Taking into account the results of [Tamaki & Sato 84], one can again replace the original program by an equivalent new one. Although equivalent, the order in which computations are done is different and some termination problems could be created. See [Debray 88] for a discussion of this issue.

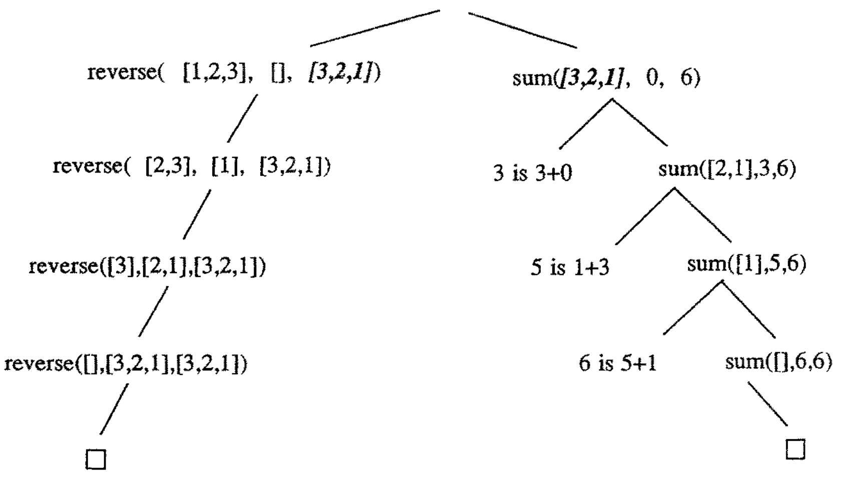

reverse([1,2,3], [], *[3,2,1]*)
reverse([2,3], [1], [3,2,1])
reverse([3],[2,1],[3,2,1])
reverse([],[3,2,1],[3,2,1])
□
sum(*[3,2,1]*, 0, 6)
3 is 3+0
sum([2,1],3,6)
5 is 1+3
sum([1],5,6)
6 is 5+1
sum([],6,6)
□

Fig.4: The toplevel calls to reverse and sum share the list [3,2,1], however, no sharing of corresponding parts occurs at the lower levels.

4. Compiling control and its relation to explanation-based learning

In the previous sections we illustrated the fact that an example can be quite useful in guiding various program transformation schemes. In this section, we discuss a related technique, which lies between program transformation and explanation-based learning (EBL). It is closely related to program transformation in the sense that it includes operations such as unfolding, the introduction of new predicates (eureka) and folding. In fact, it is a variant of an existing program transformation method called *compiling control* introduced in [Bruynooghe et al 86] (see [Bruynooghe et al 89b] for

a more detailed discussion). However, the variant that we will introduce here cannot be classified as a transformation method, because the new rules that it derives are, in general, not complete with respect to the original program. As in EBL, the new rules are added to the program in order to provide more efficient problem solving capacities for a particular subclass of queries.
Apart from this observation, the technique's relationship to EBL lies mostly in the overall strategy which is applied. Similar to the EBL-strategy, we can identify three basic steps : First, the computation which arises for a particular example query, following an efficient proof-procedure, is traced. Next, we generalize the observed computation trace (by generalizing all the concrete input given in the example-query). Finally, we synthesize a set of new rules from the generalized computation trace. The main differences from EBL are:

1/ EBL is only concerned with the logical contents of the rules. It disregards the inference strategy (in a logic programming environment : the computation rule) which will be applied to the rules. In the technique presented in this section, the aim is to construct the new rules in such a way, that if they are executed under a simple inference engine (in our case : the depth-first, left-to-right computation rule of Prolog), then the computation they give rise to is an imitation of a much more efficient proof strategy. Thus, procedural aspects will play an important role in the technique. This is reflected in the fact that we start off from computation traces instead of prooftrees.

2/ The synthesis phase (third phase) will be more complicated in the new technique, since it includes the generation of eurekas and folding. This is a form of structural generalization in the sense of [DeJong 88]. With it, we usually obtain more than one rule for a given example. Also, in most cases, this leads to new rules with more recursion and which often are more generally applicable.

3/ Because of the increased complexity in the synthesis step, the information included in the 'generalized' computational trace will be slightly different from the EBL-approach. In fact, we do not work with a generalization, but with an abstraction of the example trace.

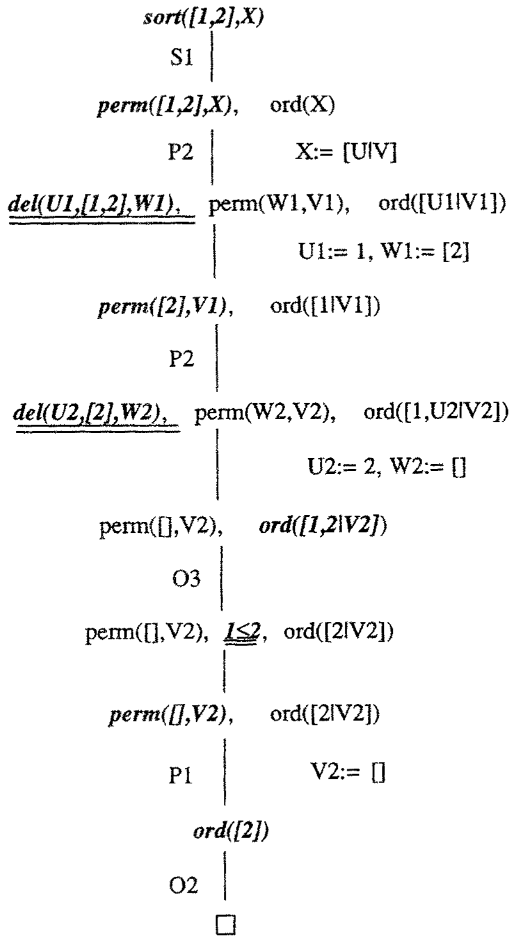
We now explain the different stages in more detail, using the permutation-sort program as an example.

```
S1:  sort(X,Y):- perm(X,Y), ord(Y).
P1:  perm([],[]).
P2:  perm([X| Y],[U| V]):- del(U,[X| Y], W), perm(W,V).
D1:  del(X,[X| Y],Y).
D2:  del(U,[X| Y],[X| Z]):- del(U,Y,Z).
O1:  ord([]).
O2:  ord([X]).
O3:  ord([X,Y| Z]):- X≤Y, ord([Y| Z]).
```

This program implements a very inefficient sorting algorithm if it is executed under the (standard) depth-first, left-to-right computation rule. It sorts by first generating a permutation of an input-list and then checking whether the permuted list is ordered. A much more efficient (although certainly not optimal) algorithm can be obtained by changing the computation rule. By coroutining the predicates perm and ord we obtain that as soon as an additional member of the permuted list is generated, the corresponding call to the ord-predicate is activated to check whether the list remains ordered. With this computation rule, failing branches in the prooftree are detected much sooner, thereby improving the efficiency.
The idea behind the method is to observe how the original rules behave under the coroutining computation rule for a particular example-query (say sort ([1,2],X)), and then, to derive new rules (which are correct in general and complete for the example) which imitate the observed behaviour if they are executed under the standard computation rule.
A trace of the computation obtained for sort ([1,2],X) under the coroutining computation rule is

displayed in Fig.5. The nodes in the tree are the consecutive resolvents derived from the original goal. The arcs are labeled with :
- the effect of the most general unifier (mgu) computed in the derivation step on the variables in the previous resolvent,
- an identification of the clause which was used in the derivation step.

The user interactively specifies the desired computation rule by selecting from the resolvent the subgoal which should be further expanded. In Fig.5 the selected subgoals are denoted in bold (e.g.



sort([1,2],X)
S1
perm([1,2],X), ord(X)
P2
X:= [U|V]
del(U1,[1,2],W1), perm(W1,V1), ord([U1|V1])
U1:= 1, W1:= [2]
perm([2],V1), ord([1|V1])
P2
del(U2,[2],W2), perm(W2,V2), ord([1,U2|V2])
U2:= 2, W2:= []
perm([],V2), ord([1,2|V2])
O3
perm([],V2), 1≤2, ord([2|V2])
perm([],V2), ord([2|V2])
P1
V2:= []
ord([2])
O2
□

Fig.5: The (concrete) computation trace tree for sort([1,2],X).

perm *([1,2],X)*, in ***perm****([1,2],X)*, ord(X)). It frequently occurs that the subgoal selected from the resolvent is not in need of any further control-directives (because the subgoal already behaves efficiently under the standard computation rule). If this is the case, the user may specify such information to the system. The subgoal will not be expanded for just one derivation step; it will be completely solved using the standard computation rule. The part of the trace corresponding to such a subcomputation is omitted. These predicates correspond to operational predicates in EBL.
In Fig.5, every call to the predicate del is of this type. It is denoted in the trace by a double

underlining of the selected subgoal.
We conclude the discussion on this first phase of the technique by observing that, although in our example there is only one branch in the tree, in general the trace may take the form of an OR-tree, representing different alternative solutions to the example query.

In a second step of the method, we convert the concrete trace into an abstract one. As in EBL, the main purpose is to generalize all information which is directly related to the example. The reason why we use abstraction instead of generalization is that we want to keep the information regarding the instantiations (e.g. ground or free) obtained at every depth within the trace explicitly represented. Originally, the method was designed to 'compile' a certain type of computation rules, in which the subgoal selection is exclusively based on the instantiation patterns of the subgoals (e.g. ***ord ([1,2],V2)*** is selected because it is sufficiently instantiated in order to perform one $\leq$-test). We call these rules *instantiation based computation rules* (see [De Schreye & Bruynooghe 89a] for a formal definition). Since the instantiation patterns form the basis of the users selections, they are included in the (generalized) trace as essential input to the synthesis algorithm.
Concretely, the abstract trace tree is obtained from the concrete one by replacing every ground term by the character g. Also, substitution-labels are removed from the arcs. This information was redundant anyway, because all unifications are implicitly represented by means of the clause-identifications labeling the arcs. The resulting abstract trace for sort ([1,2],X) is shown in Fig.6.

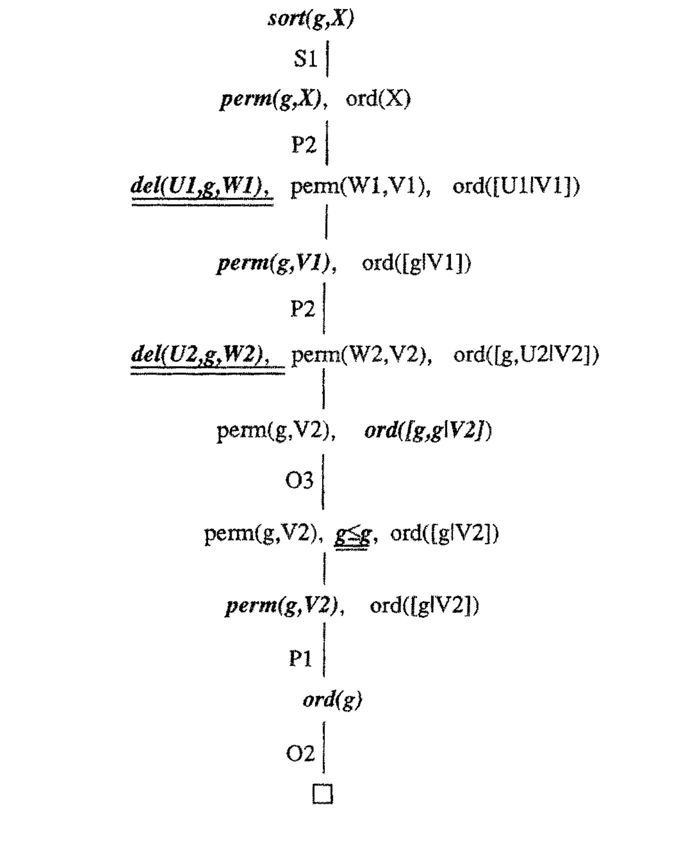

sort(g,X)
S1
perm(g,X), ord(X)
P2
del(U1,g,W1), perm(W1,V1), ord([U1|V1])
perm(g,V1), ord([g|V1])
P2
del(U2,g,W2), perm(W2,V2), ord([g,U2|V2])
perm(g,V2), ord([g,g|V2])
O3
perm(g,V2), g≤g, ord([g|V2])
perm(g,V2), ord([g|V2])
P1
ord(g)
O2
□

Fig.6: Abstract computation trace associated to the example sort([1,2],X).

The synthesis algorithm is by far the most complex component of the technique. In EBL, we would cover the entire tree with one new rule. Assuming that the predicate del is declared operational and that the calls solved with a fact are considered as being operational as well, then this new rule is of the form:

S2: sort([X,Y| Z],[U,V| W]):- del(U,[X,Y| Z],[P| Q]), del(V,[P| Q],R), U≤V,
perm(R,W), ord([V| W]).

In this clause, the atoms in the body have been ordered according to the sequence in which they were expanded (or solved) in the trace tree. The main disadvantage with this new rule is that, although it 'compiles' the desired, efficient computation rule for the generation and the testing of the first two members of the sorted list, it falls back on the original sorting algorithm afterwards.
A program transformation approach to deal with this, is to attempt a folding on the atoms del(V,[P|Q],R) and perm(R,W) using the rule P2 to obtain perm([P|Q],[V|W]), and then to fold on the atoms perm([P|Q],[V|W]) and ord([V|W]) using the rule S1 to obtain sort([P|Q],[V|W]). Unfortunately, with this folding operation we have changed the computation rule again and the new rule:
sort([X,Y| Z],[U,V| W]):- del(U,[X,Y| Z],[P| Q]), U≤V, sort([P| Q],[V| W]).

will fail due to a runtime error caused by the call to U≤V.
The alternative is to first add a redundant call to del(V,[P| Q],R) in the body of S2 immediately following the call to U≤V. With the same two foldings as described above, we then obtain:
sort([[X,Y| Z],[U,V| W]):- del(U,[X,Y| Z],[P| Q]), del(V,[P| Q],R), U≤V, sort([P| Q],[V| W]).

If, however, we want to avoid the redundant call to the del-predicate, then we need a method in which we generate more than one rule. In the example, we need at least one rule to cover the initialization (performing an initial del-operation) and another rule to deal with the recursion (an additional del-operation and the ≤-test). This is essentially what the final result of our synthesis algorithm will be.
Quite opposite to EBL, we initially generate one new rule for each derivation step in the trace. Later on, we fit the different rules together, so that their combination will mimic the recursion observed from the trace. Finally, we perform inline expansion to produce an equivalent, but minimal, set of rules.
Synthesizing the individual derivation steps from the trace is not hard. As an example, the derivation

$$\begin{tikzcd} \boldsymbol{\mathit{perm(g,X)}},\ ord(X) \arrow[d, "P2"'] & (state\ \ 2) \\ \underline{\underline{\boldsymbol{\mathit{del(U1,g,W1)}}}},\ perm(W1,V1),\ ord([U1|V1]) & (state\ \ 3) \end{tikzcd}$$

using the clause:

P2: perm([X1| Y1],[U1| V1]):- del(U1,[X1| Y1],W1), perm(W1,V1).

can be synthesized with the clause:

perm([X1| Y1],[U1| V1]), ord([U1| V1]):- del(U1,[X1| Y1],W1), perm(W1,V1), ord(([U1| V1]).

However, this is not a Horn clause. We can convert it into a Horn clause by introducing two meta-predicates, say Q and R, and instead generating the clause:

Q(perm([X1|Y1],[U1|V1]),ord([U1|V1])):-R(del(U1,[X1|Y1],W1),perm(W1,V1),ord(([U1|V1])).

If during the execution of the new set of rules a resolvent is obtained of the type

Q(perm(g,X), ord(X)) (~ state 2)

then the application of the above clause will yield a new resolvent of the type

R(del(U1,g,W1), perm(W1,V1), ord([U1|V1])) (~ state 3)

where the unifications will be identical to the ones produced by applying P2 to a resolvent of type state 2.
Similarly, for a derivation such as:

$\underline{\underline{\textbf{\textit{del(U1,g,W1),}}}}$ perm(W1,V1), ord([U1|V1]) (state 3)

|

perm(g,V1), ord([g|V1]) (state 4)

where the selected subgoal is completely solved using the standard computation rule, we introduce a clause

R(del(U1,X,W1),perm(W1,V1),ord([U1| V1])):- del(U1,X,W1), T(perm(W1,V1),ord([U1| V1])).

The remaining problem is to link the different clauses together, so that the recursive patterns implicitly contained in the trace will be integrated in the new program. To do this, we apply the following steps:

1. We partition the set of resolvents in the trace into equivalence classes. The equivalence relation must be such that equivalent resolvents give rise to 'similar' continuations of the computation in the trace.
2. All the resolvents in a same equivalence class obtain the same meta-predicate.
3. We partition the set of all derivation steps in the trace into equivalence classes. Here, equivalent derivations should be such that they can be synthesized by the same new clause.
4. We synthesize a new clause for each equivalence class of derivations, using the method described above.

A simple equivalence relation on the set of resolvents is: state i is equivalent to state j if the same set of instantiated atoms (= predicates + their instantiations) occurs in both states. If the new computation rule is consistent, then the same instantiation atom will be selected for expansion or solving from equivalent states. In this sense, the definition captures an element of recursive behaviour contained in the trace. With this equivalence relation, the trace of Fig.6 contains only one pair of equivalent resolvents, namely:

perm(g,V1), ord([g|V1]) (state 4)

and:

perm(g,V2), ord([g|V2]) (state 8)

All other equivalence classes are singletons.
We introduce the meta-predicates in the way it was illustrated in the examples above. For each of the 8 equivalence classes, we have a new predicate, say Q1, Q2,...,Q8. The arity of each predicate Qi is equal to the number of distinct instantiation atoms occurring in each of the members of its class. The subgoals in the resolvents become arguments in the new predicate. (This is a simplification of what needs to be done in the general case. Since in principle, a resolvent may very well contain more than one subgoal of a given instantiation atom, we may have more subgoals in the resolvent than there are argument positions in the new predicate. In such a case, subgoals of the same type are grouped into a list and the list will obtain an argument position in the meta-predicate.

See [De Schreye & Bruynooghe 89a] for more details).
A good way to define the equivalence of two derivation steps is :
1. both derivations start from equivalent states,
2. both derivations end up in equivalent states,
3. the clause which was expanded (or the subgoal which was solved) in the derivations is identical.

It is not hard to see that if two derivations are equivalent by the above definition, then they can be synthesized by the same new rule. For the derivations of Fig.6 we see that every equivalence class is a singleton. This is due to the example sort([1,2],X). Starting the entire process with the example sort([1,2,3],X) would yield the same number of equivalence classes, although the number of derivation steps would be increased.
Finally, using the synthesis procedure which was illustrated before, we obtain the following set of new rules :

```
Q1(sort(X,Y)):-  Q2(perm(X,Y),ord(Y)).
Q2(perm([X|Y],[U|V]),ord([U|V])):-  Q3(del(U,[X|Y],W),perm(W,V),ord([U|V])).
Q3(del(U,X,W),perm(W,V),ord([U|V])):-  del(U,X,W), Q4(perm(W,V),ord([U|V])).
Q4(perm([X|Y],[U|V],ord([U1,U|V])):-  Q5(del(U,[X|Y],W),perm(W,V),ord([U1,U|V])).
Q5(del(U,[X|Y],W),perm(W,V),ord([U1,U|V])):- del(U,[X|Y],W), Q6(perm(W,V),ord([U1,U|V]))
Q6(perm(W,V),ord([U1,U|V])):-  Q7(perm(W,V), U1≤U, ord([U|V])).
Q7(perm(W,V), U1≤U, ord([U|V])):-  U1≤U, Q4(perm(W,V),ord([U|V])).
Q4(perm([],[]),ord([U|V]):-  Q8(ord([U|V])).
Q8(ord([X])).
```

Clearly, the number of clauses can easily be reduced by using inline expansion. With the exception of Q1 and Q4, every predicate occurs twice : once as a call and once as the head of a clause. By performing the inline expansion (and by dropping the Q1-predicate) we obtain the final set of new rules :

```
sort([X|Y],[U|V]):-  del(U,[X|Y],W), Q4(perm(W,V),ord([U|V])).

Q4(perm([X|Y],[U|V],ord([U1,U|V])):-  del(U,[X|Y],W), U1≤U, Q4(perm(W,V),ord([U|V])).
Q4(perm([],[]),ord([X])).
```

In [De Schreye & Bruynooghe 89a] it is proved that the synthesis technique is correct in the sense that any answer substitution computed by the new rules under the standard computation rule - and this for any query with the same instantiation pattern as the example (in this case sort(X,Y) with X ground and Y free) - is also computed by the original program, if it is executed under the new computation rule.
Completeness of the new rules cannot be guaranteed. However, in our experiments we observed that the program obtained from one simple example usually covers a large class of other queries (with the same instantiation pattern) as well. In the case of permutation-sort, the new rules derived for sort ([1,2],X) will correctly (and much more efficiently) sort any non-empty input list. Moreover, the technique is especially well suited for incremental program development. The new rules for permutation-sort can be interactively tested using various new example queries. If a query is found for which the new rules do not compute (all) the expected answer(s), it is fed back to the system. Then, the concrete and the abstract trace for the new example are constructed. The new abstract tree is superimposed over the old one. For sort([1,2],X) and sort([],X), the initial part of the resulting combined abstract trace tree is shown in Fig.7. Finally, the synthesis algorithm is reactivated. After inline expansion, it now produces four rules : the three rules from the previous synthesis, and in addition, the rule :

```
sort([],[]).
```

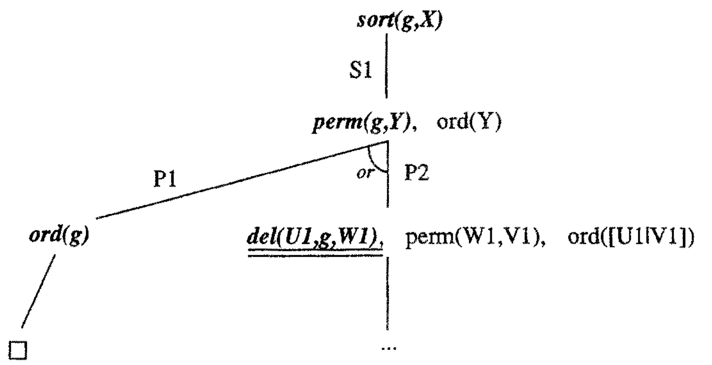


Fig.7: Fragment of the combined abstract trace for the queries sort([1,2],X) and sort([],X).

The program obtained from these two examples is complete. More generally, as an easy consequence of a completeness theorem in [De Schreye & Bruynooghe 89a], we have:

"For every given program, query instantiation pattern and instantiation based computation rule, there exists a finite number of example queries of the given pattern, such that the set of new rules synthesized from the combined abstract trace obtained for these examples is complete with respect to the original program executed under the new computation rule."

In most EBL-systems, each new example leads to a new rule. This has the disadvantage of greatly increasing code size and it can also have a negative impact on the performance of the rules (as illustrated in [Minton 88]. In [Sablon et al 89], it is shown how different rules can be combined into one. The method it proposes for synthesizing parts of two proof trees by the same new rule is very similar to our definition of equivalent derivation steps. The main reason why we can still obtain more general (and often complete) sets of rules, is that we are compiling a new (more efficient) computation rule. Therefore, we are much less confronted with the trade-off of operationality versus generality.

5. Discussion

Our aim in this paper was to explore the relationship between program transformation and explanation-based learning. Also, we wanted to illustrate how both techniques can benefit by borrowing ideas from the other. There are two main observations which form the basis for the relationship between the techniques:

1. As was pointed out in [Van Harmelen & Bundy 89], explanation-based learning and partial evaluation basically use the same technique, namely unfolding. However, they differ in the way they control the unfolding (especially, when to terminate). Here, explanation-based learning offers a new solution by using the prooftree of an example to decide on the depth of the unfolding.
2. Partial evaluation is itself an instance of the much more general unfold-fold transformation technique [Burstall & Darlington 77]. Because of the increased generality, the automatic generation of an appropriate control is even more difficult for unfold-fold than for partial evaluation. On the other hand, unfold-fold provides a more extensive set of basic transformations (e.g. eureka, folding), so that it can often produce better programs.

These observations suggest two approaches for a further cross-fertilization of the two techniques. One is the use of the prooftree as an example to guide the unfold-fold transformation, with the aim of solving the control problem. The second is to integrate a larger subset of basic unfold-fold transformations within explanation-based learning, in order to improve the operationality of the derived rules. In this paper, we focussed mainly on the first of these approaches. However, through our examples, we also gave some implicit suggestions which, we hope, will initiate further work on the second.

The explanation-based program transformation, discussed in sections 2 and 3, clearly borrows ideas from explanation-based learning, since it includes a form of example-guided unfolding. In addition, it also realizes example-guided folding and it can introduce new predicates. Consequently, it can modify the structure of the prooftree. In explanation-based learning, Shavlik and DeJong ([Shavlik & DeJong 87a,b]) have also developed a method which restructures the prooftree in case of repeated application of the same rule.

In the explanation-based transformation technique, we have borrowed ideas from program transformation to eliminate clauses while maintaining completeness. Traditional explanation-based learning 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 proofs of equivalence between sets of rules. In explanation-based learning this could be used to remove old rules with a bad performance.

The method for the compilation of computation rules (compiling control), described in section 4, was originally designed as a program transformation technique [Bruynooghe et al 89b]. In the original version of the technique, instead of constructing a computation trace for an example and subsequently abstracting this trace, we started from an abstract query pattern and applied a mechanism for *abstract interpretation* [Bruynooghe 89] to it. Thus, we obtained the abstract trace in a single step (see [De Schreye & Bruynooghe 89b] for details). The main advantage of such an approach is that it is better suited to produce a constructive proof of the completeness of the new rules (in general, the abstract interpretation covers more alternative solutions than the example computation). A major disadvantage is that the abstract interpretation scheme is computationally much more expensive.

Both explanation-based program transformation and compiling control are closely related to unfold-fold. In fact, the unfold-fold transformation method has served as a general framework for almost every source-level transformation technique proposed for logic or functional programs. The main advantage of the method is its wide range of applications, including the introduction of tail-recursion, the avoidance of redundant computations, loop merging, partial evaluation and the compilation of control. Closely related to this advantage, is its major drawback: as we already stated, the method is hard to automate. In general, the degree of automation obtained in any implementation is inversely proportional to the size of the class of transformations it can deal with. Systems designed to support a large class of transformations (e.g. [Gregory 80], [Feather 82], [Sato & Tamaki 84]) therefore depend on some form of user-provided control.

It is in this respect that explanation-based program transformation and compiling control are quite similar. By focussing on a more specific class of transformations and by using an example to guide the process, they can both be fully automated. A distinguishing feature between them is the sequential strategy in which the three basic operations: unfolding, the introduction of new predicates and folding, are performed. In explanation-based program transformation, these three basic operations are interleaved (as it is classically done in unfold-fold). In compiling control all the unfolding operations are grouped into one sequence (building the computation trace); consecutively, all the eurekas are introduced (the generation of the new predicates Qi); and finally, all the folding is performed (the synthesis of the new rules). This strategy bears more resemblance with that of partial evaluation and explanation-based learning.

We conclude by stating the main relevance of this work for explanation-based learning. From the examples we have shown, it seems that in many cases truly operational predicates can only be obtained by restructuring the prooftree or by performing structural generalization on the computation trace. In general, this may be quite hard to do. However, in the presence of an example, a sufficient condition to trigger a particular type of restructuring or generalization can easily be formulated and the appropriate, corresponding synthesis procedure can be completely predicted.

REFERENCES

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

[Bruynooghe 89] Bruynooghe M., A practical framework for the abstract interpretation of logic programs, Journal Logic Programming, 1989, to appear.

[Bruynooghe et al 86] Bruynooghe M., De Schreye D. and Krekels B., Compiling Control, in Proc.Third International Symposium on Logic Programming, 1986, pp.70-78.

[Bruynooghe et al 89a] Bruynooghe M., De Raedt L., De Schreye D., Explanation-based program transformation, in Proc. International Joint Conf. Artificial Intelligence (IJCAI89), 1989, to appear.

[Bruynooghe et al 89b] Bruynooghe M., De Schreye D. and Krekels B., Compiling Control, Journal Logic Programming, 1989, pp: 135-162.

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

[Chang & Lee 73] Chang C., Lee R.C., Symbolic Logic and Mechanical Theorem Proving, Academic Press Inc., 1973.

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

[Debray 88] Debray S.K., Unfold/fold transformations and loop optimisation of Logic Programs, in Proc. SIGPLAN'88 Conf. on Programming Language Disign and Implementation, SIGPLAN Notices, Vol.23, No.7, July 1988, pp. 297-307.

[DeJong 88] DeJong G., Some thoughts on the present and future of explanation-based learning, in Proc. European Conf. on Artificial Intelligence, (ECAI88), 1988, pp. 690-698.

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

[De Schreye & Bruynooghe 89b] De Schreye D., Bruynooghe M., An application of abstract interpretation in source level program transformation, in Programming Language Implementation and Logic Programming, Deransart, Lorho, Maluszynski, eds., LNCS 348, Springer-Verlag, 1989, pp. 35-58.

[Feather 82] Feather M.S., A system for assisting program transformation, ACM Trans. Prog. Lang. and Systems 4, 1, Jan. 1982, pp. 1-20.

[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 S.T., McCarthy L.T., Explanation based generalization as resolution theorem proving, in Proc. 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 S.T., Explanation-based generalization : a unifying view, Machine Learning, Vol.1, No.1, 1986, pp. 47-80.

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

[Sablon et al 89] Sablon G., De Raedt L., Bruynooghe M., Generalizing multiple examples in explanation-based learning, in Proc. of International Workshop on Analogical and Inductive Inference, (AII89), LNCS, Springer-Verlag, to appear.

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

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

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

[Tamaki & Sato 84] Tamaki H., Sato T., Unfold/fold transformations of logic programs, in Proc. Second International Conference on Logic Programming, 1984, pp. 127-138.

[Van Harmelen & Bundy 89] Van Harmelen F., Bundy A., Explanation Based Generalization = Partial Evaluation, Artificial Intelligence, Vol.36, No.3, 1988, pp. 401-412.

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

[Wadler 84] Wadler P., Listlessness is better than laziness, lazy evaluation and garbage collection at compile-time, in Proc.ACM Symp. on Lisp and Functional Programming, 1984, pp. 45-52.

[Wadler 85] Wadler P., Listlessness is better than laziness II: composing listless functions, in Proc. Conf. on Programs as data objects, Gansinger and Jones, eds., LNCS, Springer-Verlag, 1985.

[Wadler 88] Wadler P., Deforestation: transforming programs to eliminate trees, in Proc. Second European Symposium on Programming (ESOP88), ed. Gansinger, LNCS, Springer-Verlag, 1988, pp. 344- 358.