Compiling bottom-up and mixed derivations into top-down executable logic programs — Source link

Journal of Automated Reasoning 7: 337–358, 1991.
© 1991 *Kluwer Academic Publishers. Printed in the Netherlands.*

Compiling Bottom-up and Mixed Derivations into Top-down Executable Logic Programs*

DANNY DE SCHREYE[‡], BERN MARTENS[§], GUNTHER SABLON, and MAURICE BRUYNOOGHE[§]
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3030 Heverlee, Belgium

(Received: 3 October 1989; accepted 5 December 1989)

Abstract. We present a technique for the compilation of bottom-up and mixed logic derivations into PROLOG-programs It is obtained as an extension of a program transformation technique called *Compiling Control*. We illustrate its applications in three different domains: solving numerical problems, integrity checking in deductive databases and theorem proving The aim is to obtain efficient PROLOG programs for problems in which a non-top-down control is most appropriate.

Key words. Control rules, transformation, logic programming.

1. Introduction

Compiling Control is a program transformation technique for pure logic programs proposed in [1]. It was designed for compilation of non-standard computation rules. More specifically, given a logic program, a query pattern of interest and an (ideal) computation rule for the program and query pattern, a new logic program is derived, such that, if it is executed under the standard computation rule of PROLOG, then the computation obtained is a precise imitation of the behaviour of the original program under the ideal computation rule. Typically, such a transformation can compile a coroutining rule for any given generate-and-test program, without the use of a delay predicate. It can also compile more complex control regimes suitable for solving constraints problems, such as *forward checking* or *looking ahead* (see [2, 3]).

The technique consists of two separate steps. First, it constructs a finite fragment of an abstract computation trace obtained by executing an abstract query (the given query pattern) under the new (top-down) computation rule. In a second step, it synthesizes a new logic program which behaves as described in the trace fragment if it is executed under the standard computation rule.

*Work partly supported by ESPRIT BRA COMPULOG (project 3012).
[‡]Supported by the Belgian I W.O.N.L.-I R.S.I.A under contract number 5203. Author for correspondence.
[§]Supported by the Belgian National Fund for Scientific Research.

The starting point for this paper is the observation that we may omit the condition that the trace fragment must be obtained from a top-down execution. This provides the possibility of compiling bottom-up and mixed computation strategies.

The range of potential applications for such a transformation technique seems large. First, there are applications in solving numerical problems. Ref. 4 describes a different program transformation method aiming at the elimination of redundant computations. In all the examples we have studied so far, programs which give rise to redundant computations under a top-down execution strategy (e.g., computing the Fibonacci numbers) are more naturally and more efficiently executed with a bottom-up strategy. We use an example (Fibonacci) from this class of problems to introduce our technique.

We also illustrate how another transformation technique, namely the transformation of *almost-tail-recursive* procedures into logically equivalent *tail-recursive* ones (see [5]), can be imitated by compiling a bottom-up control rule, to obtain equally efficient tail-recursive procedures.

A quite different class of applications is the compilation of integrity checking in deductive databases. Refs. 6 and 7 present a technique based on theorem proving, using mixed inference strategies to check the integrity constraints. In [8] an implementation based on a meta-interpreter is described. In this paper, we will briefly illustrate how our technique can be used to enable a compiled approach.

Finally, there are applications in theorem proving. If a (resolution based) theorem prover can produce a successful derivation path for a given type of problem, expressed in (not necessarily Horn-) clausal formulas, then our technique can compile the derivation path into an efficient logic program. In a learning environment, this could provide a practial tool for deriving operational rules, similar to what is achieved by *Explanation-based Learning* [9, 10] but for problem domains which are specified in full, non-Horn, clausal form.

Before going in more detail, we want to emphasize that what we are presenting is a technique for control compilation. Given a clausal theory, a query of interest and an *ideal control rule* for this theory and query, we derive a PROLOG program that compiles the triplet. The problem of how to obtain the ideal (or even a good) control rule is not addressed. We will assume that it is either provided by the user or that it is obtained from a general theorem prover. For some of the specific applications we discuss (e.g., integrity checking), we refer to some known principles for obtaining a useful strategy.

2. Compiling Control Rules

In this section we introduce our technique using two simple examples. The first one describes the compilation of a non-standard top-down computation rule. In the context of this example, we recall a subset of the control compilation of [1]. Moreover, we slightly reformulate the compilation procedure, so that we can easily extend it do deal with non-top-down computations. The second example describes the compilation of a bottom-up control rule.

Our notational conventions are variables, function- and predicate names start with a lowercase character. Upper case is used for constants. With the infix dot-notation, $x.y$, we represent a list with head x and tail y.

The first example is the transformation of the permutation-sort program. The original clauses are

S1: sort(x, y) $\leftarrow$ perm(x, y), ord(y).
P1: perm(Nil, Nil)$\leftarrow$.
P2: perm($x.y$, $u.v$) $\leftarrow$ del(u, $x.y$, w), perm(w, v).
D1: del(x, $x.y$, y)$\leftarrow$.
D2: del(u, $x.y$, $x.v$) $\leftarrow$ del(u, y, v).
O1: ord(Nil)$\leftarrow$.
O2: ord(x.Nil)$\leftarrow$.
O3: ord($x.y.z$) $\leftarrow$ $x \leqslant y$, ord($y.z$).

The query pattern of interest is $\leftarrow$sort(x, y). The new computation rule is expressed by building (a finite part of) the computation trace which is obtained by executing the query $\leftarrow$sort(x, y) under the new computation rule. For permutation-sort, the computation trace is drawn in Figure 1.

The trace has the shape of an OR-tree. Its nodes contain an identifier for the node and a resolvent obtained during the derivation under the new computation rule. In each resolvent, the subgoal selected by the rule is denoted in bold-italic. This subgoal is expanded for one derivation step using every applicable clause from the program.

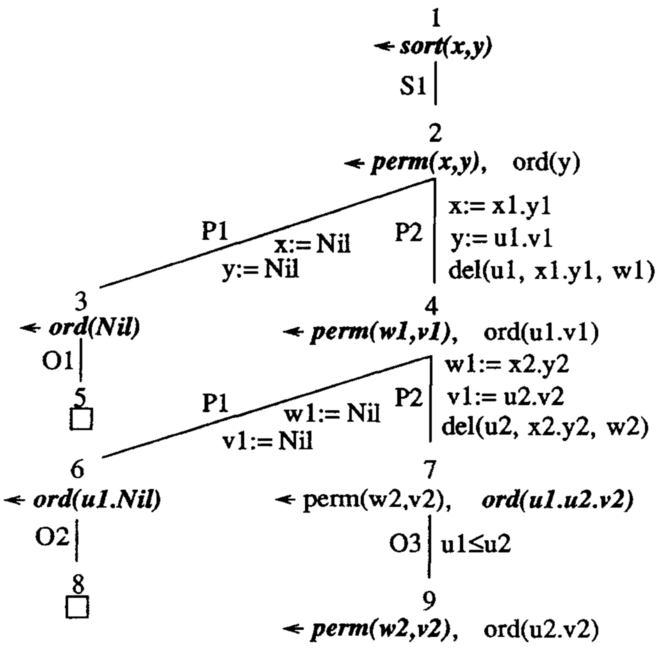

Fig. 1 Fragment of the computation trace tree for permutation-sort under a coroutining rule.

Arcs connecting the nodes are drawn to represent such derivation steps. An arc is labeled with an identifier of the clause which was used in the derivation step (left of the arc) and with the effect of the resulting most general unifier (mgu) on the variables in the father node (right of the arc).

As can be observed from Figure 1, not all consecutive resolvents are represented as nodes (e.g., ←del($u1$, $x1.y1$, $w1$), perm($w1$, $v1$), ord($u1.v1$) is the resolvent obtained by applying clause P2 to the node-2-resolvent ← perm(x, y), ord(y), and has not been represented. The reason is that some selected subgoals are dealt with in a different way. Some subgoals occurring during the computation are not in need of a non-standard control. They can be efficiently executed under the standard computation rule. All resolvents containing such subgoals have been omitted from the trace. Instead, these subgoals are represented as extra labels on the arcs of the derivation steps that created them (e.g. del($u1$, $x1.y1$, $w1$) in the derivation from node 2 to node 4). The interpretation of this feature of the trace tree is that such subgoals are immediately and completely solved using the standard computation rule. The successor node(s) in the trace represent(s) the next resolvent(s) in the derivation, which is (are) obtained after the subgoal has been completely solved.

Observe that node 4, ←perm($w1$, $v1$), ord($u1.v1$), actually represents a collection of resolvents. The goal ←del($u1$, $x1.y1$, $w1$) can succeed a number of times. For each success, a different mgu is produced (the effect of these mgu's is not represented in the trace). Node 4 represents all the resulting resolvents. This means that within the trace, the backtracking behavior is not solely expressed by the different OR-branching-points. The computation may backtrack up to the latest del/3-call, generate a new success-substitution, and redescend down the same branch, using this substitution.

In addition to predicate calls that behave efficiently under the standard computation rule, this special feature of the computation traces will also be used to deal with built-in predicates (e.g., $u1 \leqslant u2$ in Figure 1) and calls to database predicates (see Section 3).

Here, we must point out that we do not provide a general decision procedure to determine whether a goal should be expanded or completely solved. The only automated rule is that calls to built-ins and database facts are always solved. The absence of such a decision procedure may seem worrying at first sight. However, it is important to realize that the only purpose of the 'solve' feature (in the case of non-built-ins) is to reduce the size of the trace tree and the complexity of the transformation. If we would simply have expanded the goal ←del($u1$, $x1.y1$, $w1$), and its descendants with the depth-first, left-to-right computation rule, then, with a larger trace tree, we would have obtained an equivalent transformed program.

This does not mean that the feature is redundant. Its purpose is threefold: (1) calls to built-ins cannot be unfolded, (2) expansion of database calls would produce a very high number of branchings in the trace tree, (3) starting from a large knowledge base, it is essential from a practical point of view that the transformation can be focused on certain predicates of this knowledge base, which are in need of a non-standard control, leaving all others untouched.

We now turn our attention towards the compilation of the computation rule. Figure 1 represents only a finite fragment of the infinite computation trace obtainable for ←sort(x, y) since we cannot use the entire, infinite trace as input to the compilation procedure. However, we do want the new program to be a compilation for the entire trace. Therefore, we have to impose some condition on the relation between the trace fragment and the infinite trace.

Recall that we have associated a unique identifier with each node in the computation trace. We will assume that these identifiers are integers and that they increase from top to bottom in the trace and from left to right. We say that a resolvent $R1$ is previous to a resolvent $R2$ if the node-identification of $R1$ is smaller than the one of $R2$. Furthermore, the set of labels on an arc starting from a resolvent R in the trace is called an *action* prescribed by the computation rule in R. As an example, in Figure 1, the actions prescribed in the resolvent ←perm(x, y), ord(y) are $\{P1, x :=$ Nil, $y :=$ Nil$\}$ and $\{P2, x := x1.y1, y := u1.v1,$ del($u1$, $x1.y1$, $w1$)$\}$

DEFINITION 2.1. A computation rule r is *consistent* if for any resolvent $R1$ which is a renaming of the previous resolvent $R2$, the actions prescribed by r in $R1$ are the (same) renaming of the actions it prescribed in $R2$.

In the example, the resolvent in node 9, ←perm($w2$, $v2$), ord($u2$, $v2$), is a renaming of the one in node 4, ←perm($w1$, $v1$), ord($u1.v1$). If our computation rule is consistent, then the renamed actions $\{P1, w1 :=$ Nil, $v2$: Nil$\}$ and $\{P2, w2 := x3.y3, v2 := u3.v3,$ del($u3$, $x3.y3$, $w3$)$\}$ are taken in node 9. Therefore, the resolvents in node 10 and node 11 will be renamings of the ones in node 6 and node 7. By induction, every following resolvent and action is a renaming of a corresponding resolvent and action which belongs to the finite fragment. The entire infinite computation trace then folds into a finite graph. We call it the compilation graph. For permutation-sort it is shown in Figure 2.

The compilation graph is very similar to the trace fragment. It can be obtained from it by

- Omitting every (non-empty) resolvent $R1$ which is a renaming of a previous resolvent $R2$, together with all its descendants.
- Redirecting the arc leading to $R1$ in the trace fragment towards $R2$. The renaming substitution is added as an additional label to this arc.

Nodes such as $R2$ will be referred to as *loop-nodes*.

Finally, we generate a set of new Horn clauses which synthesize the compilation graph. To this purpose, we start by focusing on three particular types of nodes in the graph: the root, the nodes containing the empty clause and loop-nodes. These nodes are called *principal nodes*. We then synthesize one new clause for each path in the graph which connects two consecutive principal nodes.

A first such path leads from node 1 to node 5. We obtain the clause: $N1$: sort(Nil, Nil)←.

The three remaining paths all include the loop-node. To synthesize them, we introduce a new predicate $p/2$. This predicate is used to build a canonical meta-representation

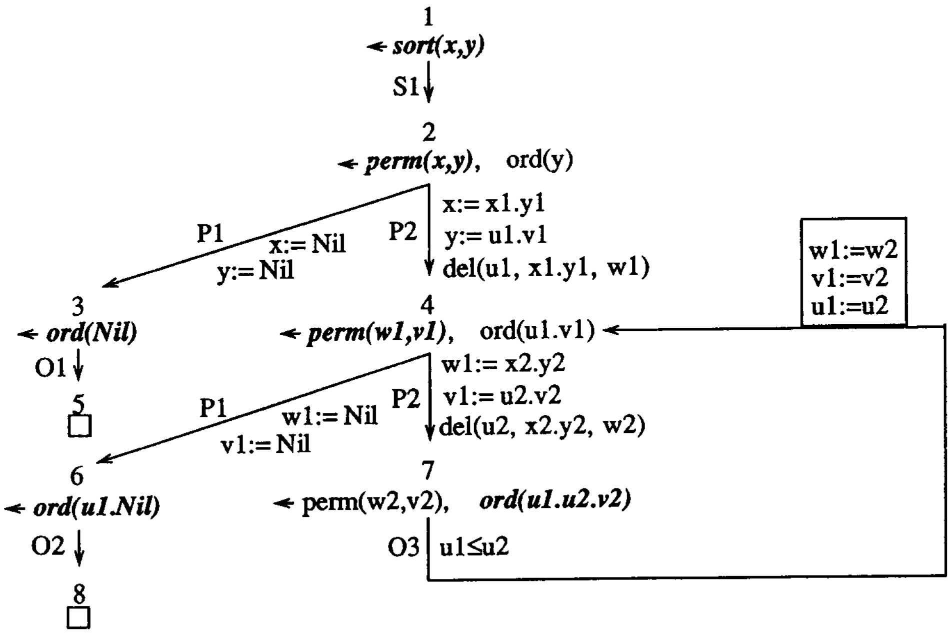

Fig. 2. The compilation graph for permutation-sort.

for the resolvent, ←perm($w1$, $v1$), ord($u1.v1$) in the loop-node. The meta-representation is ←p(perm)$w1$, $v1$), ord($u1$, $v1$)). With this new predicate and using the synthesis algorithm [11], we obtain

- for the path from node 1 to node 4:

  $N2$: sort($x1.y1$, $u1.v1$) ← del($u1$, $x1.y1$, $w1$), p(perm($w1$, $v1$), ord($u1.v1$)).

- for the path from node 4 to node 8:

  $N3$: p(perm(Nil, Nil), ord($u1$.Nil))←.

- for the path from node 4 to node 4:

  $N4$: p(perm($x2.y2$, $u2.v2$), ord($u1.u2.v2$)) ← del($u2$, $x2.y2$, $w2$), $u1 \leqslant u2$, p(perm($w2$, $v2$), ord($u2$, $v2$)).

In this last clause, the renaming substitution has been applied to the meta-representation of the loop-node's resolvent to obtain the recursive call.

We only briefly discuss the correctness and completeness of the transformation. The reader is referred to [11] for a detailed treatment.

Both the trace tree and the compilation graph represent a set of OR trees. For any given query which unifies with ←sort(x, y), a corresponding OR tree can be obtained from the computation trace by further instantiating the variables in the trace, removing the branches for which the unifications expressed in the labels fail and expanding the derivations for the subgoals solved under the standard computation rule. In the same way we obtain an OR tree from the compilation graph, by, in addition, performing unfoldings of the loops. Clearly, for each such query, there is a one-to-one correspondence respecting branching points and sequences of unifications between the two OR trees. Thus, the compilation graph correctly and completely represents the computation trace.

Next, each of the clauses $N1$ to $N4$ is precise synthesis for the corresponding path in the compilation graph. Moreover, because of the meta-predicate, the sequences (and alternatives) of clauses from $N1$ to $N4$ that are applicable for a query of the type $\leftarrow$sort(x, y) and under the standard computation rule, correspond to those in the OR tree obtained from the graph. Therefore, $N1$ to $N4$ (together with the clauses for del/3) are a correct and complete compilation of the original program under the new computation rule.

Generalizing the example, we can now formalize the compilability of a computation rule as follows.

DEFINITION 2.2. A triplet (P, q, r) consisting of a program P, a query q and a computation rule r is *compilable*, if r is consistent and if there exists a finite fragment F of the computation trace of (P, q) under r, such that for each resolvent $R1$ in a leaf of F, there exists a previous $R2$ in F which is renaming of $R1$.

We now turn our attention towards the compilation of bottom-up or mixed computations. Intuitively, the reason why this seems feasible is that from a procedural point of view any resolution based refutation is a sequence of unifications. If the different sequences of unifications occurring in a given bottom-up or mixed computation strategy can be expressed and organized within a computation trace tree, then there seems to be no objection against the application of the synthesis procedure of the previous example.

Our second example is the classical problem of computing the Fibonacci numbers. The original clauses are

F1: fib(0, 0)$\leftarrow$.
F2: fib(1, 1)$\leftarrow$.
F3: fib(n, f) $\leftarrow$ $n \geqslant 2$, plus($n1$, 1, n), plus($n2$, 1, $n1$), fib($n1$, $f1$), fib($n2$, $f2$), plus($f1$, $f2$, f).

Again, we express the desired computation strategy by means of a computation trace fragment. It is displayed in Figure 3. The fragment has three roots: the query $\leftarrow$fib(n, f) and the facts fib(0, 0) $\leftarrow$ ($F1$) and fib(1, 1) $\leftarrow$ ($F2$). We refer to the set of all the roots of a trace as the *0th layer* of the trace.

From these roots, we can either resolve on $\leftarrow$fib(n, f) and fib(0, 0)$\leftarrow$ to derive the empty clause, or we can resolve on $\leftarrow$fib(n, f) and fib(1, 1)$\leftarrow$, again obtaining the empty clause, or we can perform a bottom-up inference step, applying the rule $F3$ to the facts fib(0, 0)$\leftarrow$ and fib(1, 1)$\leftarrow$ and solving all the remaining calls in the body of $F3$ (plus(0, 1, 1), plus(1, 1, $n1$), $n1 \geqslant 2$ and plus(1, 0, $f1$)) to deduce the fact fib($n1$, $f1$)$\leftarrow$.

Each inference step may include a multiple number of operations (including more than one resolution). The three inference steps above are referred to as the *first layer*

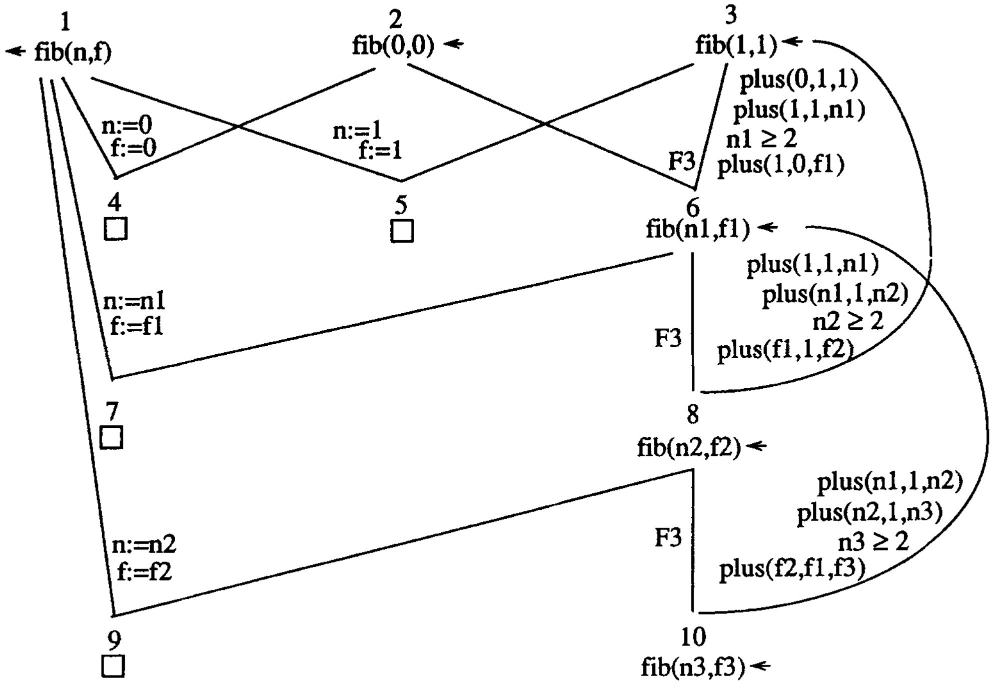

Fig. 3. Trace fragment for a mixed computation strategy for the Fibonacci numbers.

inferences of the trace. The clauses: □, □ and fib(*n*1, *f*1)← are the first layer clauses. In general, we have the following:

DEFINITION 2.3. The *0th layer clauses* of a computation trace are the roots of the trace.

An *ith layer inference step* is a sequence of resolutions *consuming* one or more $(i - k)$th layer clauses, $k > 0$, and zero or more clauses from the program. At least one of the consumed clauses is of layer $i - 1$. Each *i*th layer inference step *produces* one *ith layer clause* as its resolvent.

In Figure 3, each inference step is represented by a number of descending arcs (one arc for each clause consumed), all ending in a same node (that of the produced clause). Each node contains an identification and a clause. Clauses belonging to a same layer of the trace have been positioned on a same horizontal line. If a clause *C*1 is connected to a clause *C*2 by a path of descending arcs, we say that *C*2 is a *descendent* of *C*1.

As in Figure 1, an identifier of the clause from the program that was applied (if any) and the effect of the mgu on the variables in the parent nodes, are represented as labels on the arcs. Also, calls solved using the standard top-down control (in this example the built-ins) are labeled on the arcs.

The main difference between this trace segment and the one in Figure 1 is that this one does not have the shape of a tree. Every inference step combines information from

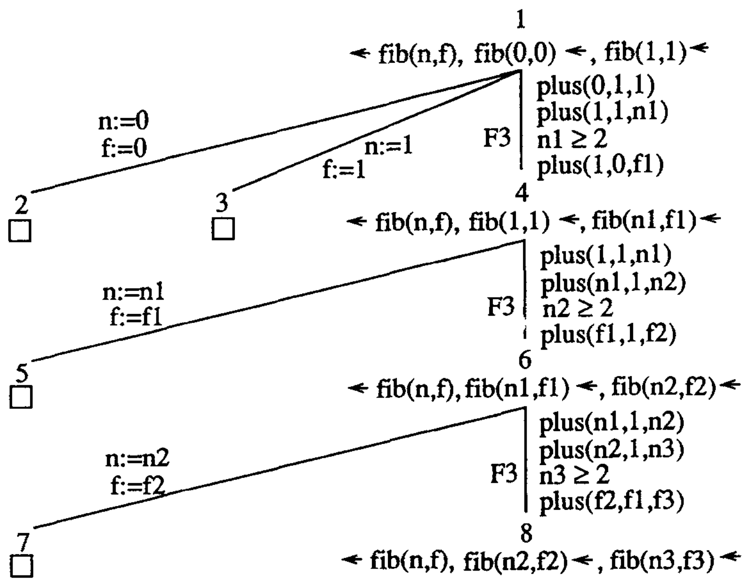

Fig. 4. Syntactical reorganization of the trace fragment for the Fibonacci numbers

different nodes in lower layers to infer the produced clause. However, the tree-structure is essential if we want to mimic the computation by means of top-down derivations.

This problem is solved by performing a syntactical reorganization on the trace fragment. A number of nodes from the initial graph are merged into one node of the new graph. The new node contains the conjunction of their clauses. This conjunction represents a computation state which exists at a given time (what is the pending query and what are the relevant facts that are available at that time). As an example, the query ←fib(n, f) and the two facts fib(0, 0)← and fib(1, 1)← form the initial computation state of Figure 4. They are merged as a conjunction into a single root of the reorganized graph, shown in Figure 4.

The following procedure computes a reorganized trace (RT) from a given computation trace (CT):

Initialization

The 0th layer of the RT consists of a single state. It is the conjunction of all 0th layer clauses of CT.

Constructing the ith layer of RT

1. The state at the ith layer:
   First we partition the ith layer clauses of CT into states. Two such clauses are conjuncts of a same ith layer state in RT if they have a common descendant at layer

$i + k, k \geqslant 1$, in CT. Let S by any ith layer state of RT obtained in this way. We add to S all the kth layer clauses R of CT, $k < i$, such that there exists a descendant D of a conjunct of S in CT and R is consumed in the inference step of CT producing D.

2. The inferences at the ith layer:
   For every ith layer state S_i of RT, there is exactly one $(i - 1)$th layer state S_{i-1}, such that there is at least one inference step of layer i in CT which consumes only conjuncts of S_{i-1} and produces a conjunct of S_i. Draw an arc from S_{i-1} to S_i. Take the union of all labels on all arc representing ith layer inferences in CT of the type described above. Add this union as a label on the arc from S_{i-1} to S_i in RT.

The main difference between a trace segment such as that of Figure 1 and the tree in Figure 4 is that the nodes of Figure 4 do not contain the resolvents of an SLD derivation. For instance, no SLD inference step can produce the effect of the transition from node 1, $\leftarrow$fib(n, f), fib(0, 0), fib(1, 1)$\leftarrow$, to the empty clause. As we will illustrate, this is not an objection to the application of the compilation procedure.

The notions of a *previous* state and of an *action* prescribed by a control rule in a state are completely similar to the corresponding notions for non-standard, top-down computation rules.

DEFINITION 2.4. A control rule r is *consistent* if for any computation state S_1, which is a renaming of a previous state S_2, the actions prescribed by r in S_1 are the (same) renaming of the actions it prescribed in S_2.

In the Fibonacci example, the computation state in node 8, $\leftarrow$fib(n, f), fib($n2$, $f2$)$\leftarrow$, fib($n3$, $f3$)$\leftarrow$, is a renaming of the previous state in node 6. Again, if the control rule is consistent, the reorganized trace folds into a compilation graph. The graph is drawn in Figure 5.

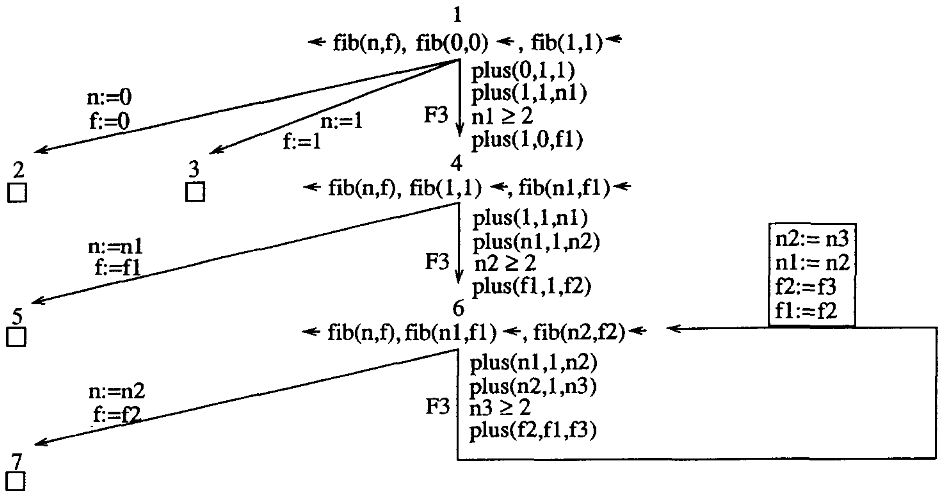

Fig. 5. Compilation graph for the Fibonacci numbers.

The synthesis of the new clauses involves only one element which is different from the synthesis for permutation-sort. Since we want to obtain a new procedure for the predicate fib/2, we add the node (0, ←fib(n, f)) as a new root on top of the compilation graph. We connect it to node 1 with an arc without labels.

Following the synthesis procedure of the previous example, we introduce a new predicate p/3 and we replace the loop-node, ←fib(n, f), fib(n1, f1)←, fib(n2, f2)← by its canonical meta-representation p(fib(n, f), fib(n1, f1), fib(n2, f2)). We synthesize clauses for paths connecting principal nodes. Three such paths connect node 0 to □. The clauses are

N1: fib(0, 0)←. (state 0 to state 2)
N2: fib(1, 1)←. (state 0 to state 3)
N3: fib(n1, f1) ← plus(0, 1, 1), plus(1, 1, n1), n1 $\geqslant$ 2, plus(0, 1, f1).
(state 0 to state 5)

The paths including the loop-node are synthesized by

N4: fib(n, f) ← plus(0, 1, 1), plus(1, 1, n1), n1 $\geqslant$ 2, plus(1, 0, f1),
plus(1, 1, n1), plus(n1, 1, n2), n2 $\geqslant$ 2, plus(f1, 1, f2), p(fib(n, f),
fib(n1, f1), fib(n2, f2)). (state 0 to state 6)
N5: p(fib(n2, f2), fib(n1, f1), fib(n2, f2))←. (state 6 to state 7)
N6: p(fib(n, f), fib(n1, f1), fib(n2, f2) ← plus(n1, 1, n2), plus(n2, 1, n3), n3 $\geqslant$ 2,
plus(f2, f1, f3), p(fib(n, f), fib(n2, f2), fib(n3, f3)). (state 6 to state 6)

The new program still contains some inefficiencies. The calls to $\geqslant$/2 which were needed in the original top-down formulation of the problem in order to ensure termination are of no use in the bottom-up compilation.

Also, every first call to plus/3 in the clauses N3, N4 an N6 is redundant. A simple theorem prover could detect these redundancies. However, we will not rely on such capabilities here.

Another problem is that the program doesn't terminate. For queries of the type ←fib(ground, any), we have one successful derivation path, after which the computation goes on indefinitely. This can easily be repaired by introducing a 'cut' at the end of each clause. However, in general it would require a determinacy analysis of the original program (see [12, 13]) or additional information from the user of the transformation system to make an appropriate decision on the insertion of 'cuts'.

Apart from these technical problems and after elimination of redundant functions in p/3, our transformation results in the most efficient implementation for the computation of Fibonacci numbers. The program strongly resembles the program which was obtained in [4] by eliminating redundant calls to fib/2. The main difference is that our program is tail-recursive.

Finally, we return to the issue of correctness and completeness of the transformation. The arguments are the same for the compilation of computation rules. The compilation graph correctly and completely represents the computation expressed in the initial (infinite) computation trace. The new clauses are a correct and complete synthesis for the compilation graph. As for compilability, we now have the following.

DEFINITION 2.5. A triplet (P, q, r) consisting of program P, a query q and a control rule r is *compilable*, if r is consistent and if there exists a finite fragment F of the reorganized computation trace of (P, q) under r, such that for each state $S1$ in a leaf of F, there exists a previous node of F, with a state $S2$, and such that $S1$ is a renaming of $S2$.

3. Applications of the Technique

The compilation can be used in a wide range of applications. In this section we illustrate its applicability in three quite different domains: solving numerical problems, integrity checking in deductive databases, and theorem proving.

3.1. SOLVING NUMERICAL PROBLEMS

When numerical problems are specified declaratively in terms of logic programs, this often results in programs which are inefficient under the standard top-down execution mechanism of PROLOG. One type of inefficiency is due to the occurrence of redundant computations (duplicate calls within a same execution). The Fibonacci problem is a typical example. In [14] several other examples of the same type are studied.

The logical specifications used by Clocksin for these problems construct a term which represents the operations needed to compute the result of the problem (e.g., $(0 + 1) + 1$ represents the operations needed for the computation of the third Fibonacci number). In Clocksin's method, common subterms of such a term are detected and then the term is folded into a graph-structure, where each subcomputation is only represented once.

Inspired by this technique Ref. 4 introduces an automatable source level transformation method, based on unfold/fold [15] and factoring. It produces new, logically equivalent logic programs, from which all redundant computations have been eliminated.

A more natural way to eliminate the redundancies is to execute the programs with a bottom-up strategy. As shown in the previous section, our compilation technique can transform fragments of such executions into top-down executable logic programs. We successfully applied the technique to all the examples discussed in [4] and [14]. These include: the approximation of the exponential function by finite series expansions, solving matrix equations, and computing an n-point discrete Fourier-transform. In general, the efficiency of our compiled programs was slightly better than in [4], because they are always tail-recursive.

We do not give an additional example of such a transformation here. Instead, we illustrate how bottom-up compilation can be used to transform almost-tail-recursive programs into logically equivalent tail-recursive ones.

Since logic programming languages have no constructs for iterative loops, such as *for* and *while*, program loops are always expressed with recursion. In terms of efficient implementation, tail-recursive procedures are a good approximation of iterative loops. However, many natural logical formulations of numerical (and other) problems are non-tail-recursive. Ref. 5 presents a transformation technique based on unfold/fold which transforms a class of recursive programs (called almost tail-recursive) into logically equivalent tail-recursive ones. The most complex example Debray deals with is the non-deterministic computation described by

$$r(x) = \begin{cases} 1 \text{ if } x = 0 \\ 2 * r(x - 1) \text{ if } x\ (>0) \text{ is even} \\ 2 * r(x - 1) - 1 \text{ or } 2 * r(x - 1) + 1 \text{ if } (x > 0) \text{ is odd} \end{cases}$$

In clausal form, the program is

*R*1: $r(0, 1) \leftarrow$.
*R*2: $r(n, f) \leftarrow n > 0$, even(n), plus($n1$, 1, n), $r(n1, f1)$, twotimes($f1$, f).
*R*3: $r(n, f) \leftarrow n > 0$, odd(n), plus($n1$, 1, n), $r(n1, f1)$, twotimes ($f1$, f).
*R*4: $r(n, f) \leftarrow n > 0$, odd(n), plus($n1$, 1, n), $r(n1, f1)$, twotimes$^+$($f1$, f).
*T*1: twotimes($f1$, f) $\leftarrow f$ is $f1 * 2$.
*T*2: twotimes ($f1$, f) $\leftarrow f$ is $(f1 * 2) - 1$.
*T*3: twotimes$^+$($f1$, f) $\leftarrow f$ is $(f1 * 2) + 1$.

Ref. 16 gives the details of the transformation. Here we show how a similar tail-recursive program can be obtained by compiling a completely bottom-up control

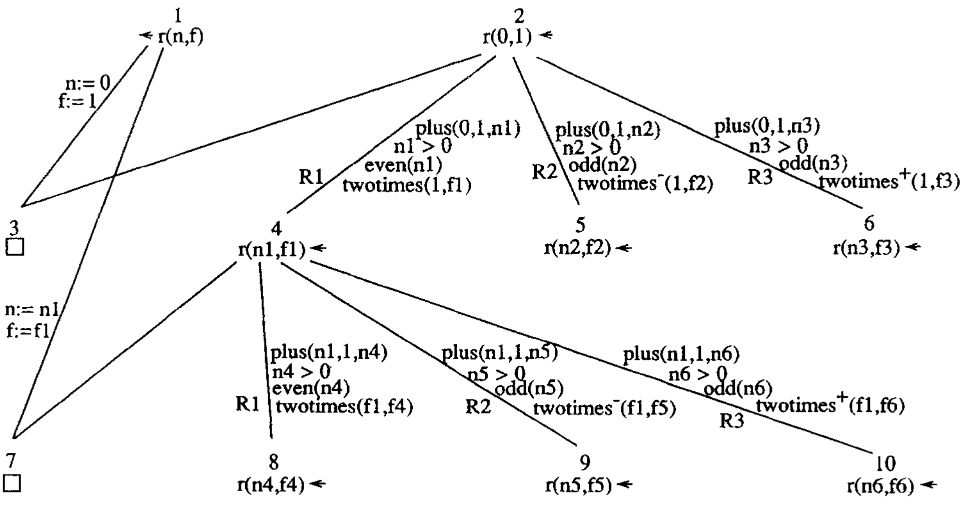

Fig. 6. Fragment of the computation trace for the bottom-up computation of the $r/2$-predicate.

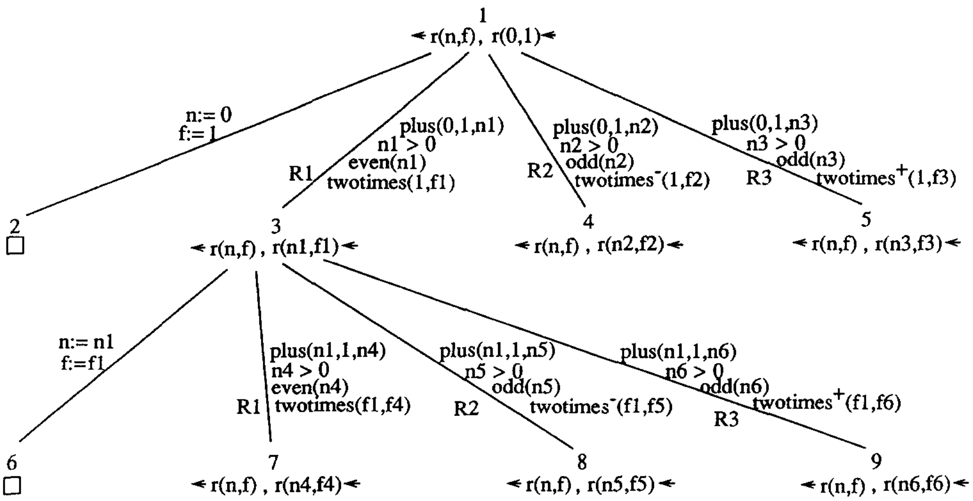

Fig. 7. Reorganized computation tree fragment for $r/2$.

rule. The computation trace fragment is drawn in Figure 6. The control rule is such that at each stage it either unifies the query $\leftarrow r(n, f)$ with a newly derived fact, or it uses the fact as input to one of the three recursive rules to derive new facts. The reorganized graph is shown in Figure 7. Each computational state is obtained by merging a fact in a node of Figure 6 with the query $\leftarrow r(n, f)$. The states in the nodes 4, 5, 7, 8 and 9 are renamings of the state $\leftarrow r(n, f)$, $r(n1, f1)\leftarrow$ in node 3. Thus, by folding on them, we obtain the compilation graph of Figure 8.

By adding the additional node (0, $\leftarrow r(n, f)$) and using the meta-predicate $p/2$ to represent loop-node 3, we synthesize the clauses:

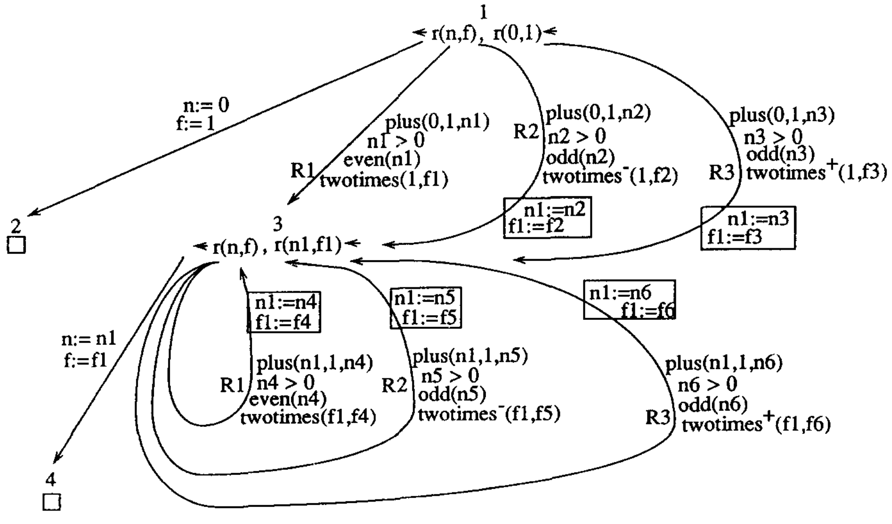

Fig. 8 Compilation graph for $r/2$.

$N1$: $r(0, 1) \leftarrow$. (state 0 to state 2)
$N2$: $r(n, f) \leftarrow$ plus$(0, 1, n1)$, $n1 > 0$, even$(n1)$, twotimes$(1, f1)$, $p(r(n, f), r(n1, f1))$. (state 0 to state 3, using $R1$)
$N3$: $r(nf) \leftarrow$ plus$(0, 1, n2)$, $n2 > 0$, odd$(n2)$, twotimes$^-(1, f2)$, $p(r(n, f), r(n2, f2))$. (state 0 to state 3, using $R2$)
$N4$: $r(n, f) \leftarrow$ plus$(0, 1, n3)$, $n3 > 0$, odd$(n3)$, twotimes$^+(1, f3)$, $p(r(n, f), r(n3, f3))$. (state 0 to state 3, using $R3$)
$N5$: $p(r(n1, f1), r(n1, f1)) \leftarrow$. (state 3 to state 4)
$N6$: $p(r(n, f), r(n1, f1)) \leftarrow$ plus$(n1, 1, n4)$, $n4 > 0$, even$(n4)$, twotimes$(f1, f4)$, $p(r(n, f), r(n4, f4))$. (state 3 to state 3, using $R1$)
$N7$; $p(r(n, f), r(n1, f1)) \leftarrow$ plus$(n1, 1, n5)$, $n > 0$, odd$(n5)$, twotimes$^-(f1, f5)$, $p(r(n, f), r(n5, f5)$. (state 3 to state 3, using $R2$)
$N8$: $p(r(n, f), r(n1, f1)) \leftarrow$ plus$(n1, 1, n6)$, $n > 0$, odd$(n6)$, twotimes$^+(f1, f6)$, $p(r(n, f), r(n6, f6))$. (state 3 to state 3, using $R3$)

The efficiency of the resulting program is of the same order as that of the transformed tail-recursive program in [16]. Clearly, it can be still further improved by trivial transformations such as the elimination of the calls to $>/2$, the introduction of 'cuts' and the removal of redundant functions in the $p/2$-predicate.

It is not surprising that the bottom-up compilation produces programs comparable to the tail-recursive programs of [5]. The technique of producing tail-recursive procedures involves the introduction of accumulating parameters. The typical function of an accumulating parameter is to simulate a bottom-up-like computation in a top-down execution.

The reader may wonder why we restrict our attention to the bottom-up compilation of numerical problems. The reason is that all recursive programs depend on an input which is either numerical or a recursively defined data-structure (or a number of database facts; a case which we discuss in the next subsection). Programs consuming input from a recursively defined data-structure are hard to execute with a bottom-up strategy. As an example, consider the program:

$S1$: scalar$(x, y, r) \leftarrow$ product(x, y, z), sum$(z, 0, r)$.
$P1$: product(Nil, Nil, Nil)$\leftarrow$
$P2$: product$(x.xt, y.yt, z.zt) \leftarrow$ is $x * y$, product(xt, yt, zt).
$SU1$: sum(Nil, r, r)$\leftarrow$.
$SU2$: sum$(x.xt, ac, r) \leftarrow a$ is $x + ac$, sum(xt, a, r).

The program computes the scalar product of two vectors, which are represented as lists. It is inefficient under a top-down execution, because the same data structure is traversed twice. A transformation merging the two program loops is given in [17]. If we want to execute this program with a bottom-up strategy, then, at each step in the recursion, we need access to the last element in the input lists (representing the two vectors to be multiplied). Therefore, in order to deal with this type of problem, we would need to extend our compilation technique with an initial phase in which every recursively defined input-data-structure is reversed. Although this seems feasible, we will not elaborate on it in this paper.

3.2. INTEGRITY CHECKING IN DEDUCTIVE DATABASES

In [6 and 7] an approach for checking integrity constraints in deductive databases, based on theorem proving, is presented. Under the assumption that the databases is consistent prior to an update (which can be the insertion or the deletion of a fact, a rule or an integrity constraint), a derivation is built with a mixed computation strategy and using the update as a generalized goal-statement. If the empty clause can be derived, then the update is clearly inconsistent with the database.

In the special case that no implicit deletions are involved in the refutation, Ref. 6 proposes a compilation scheme which compiles the computation strategy into a top-down executable program. We give a simple example to illustrate how our technique can be used to produce similar results. The initial database is

$F1$: finished(Project2)$\leftarrow$.
$S1$: subproject (Project11, Project2)$\leftarrow$.
$S2$: subproject (Project21, Project2)$\leftarrow$.
$W1$: works-on(Carl, Project1)$\leftarrow$.
$W2$: works-on(Lou, Project11)$\leftarrow$.
$W3$: works-on(x, y) $\leftarrow$ subproject(z, y), works-on(x, z).
$I1$: $\leftarrow$works-on(x, y), finished(y).

We aim to produce a logic program that checks the integrity of the database whenever a new fact of the type works-on(x, y)$\leftarrow$ is inserted. In Figure 9 a fragment of the computation trace is shown.

We chose to represent the trace in a less compact form than what we did in the previous examples (resolvents containing a subgoal which is solved completely by the standard computation rule have not been omitted) to show that the resolvents occurring in the traces are in general not restricted to facts or goals.

The trace already has the shape of a tree (this is always the case in this type of application). Therefore, no reorganization is needed. We have one loop-node,

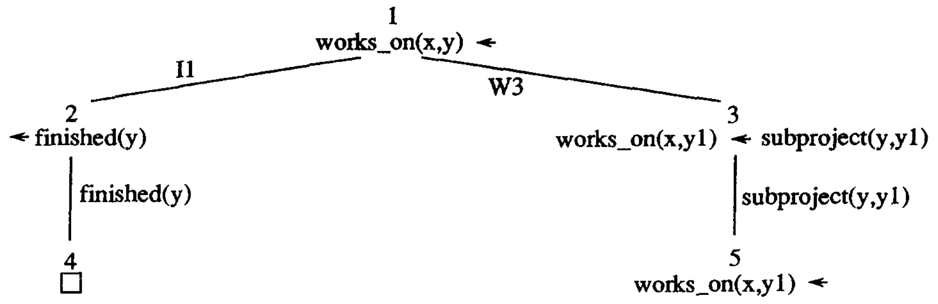

Fig. 9. Fragment of the computation trace for the works-on problem.

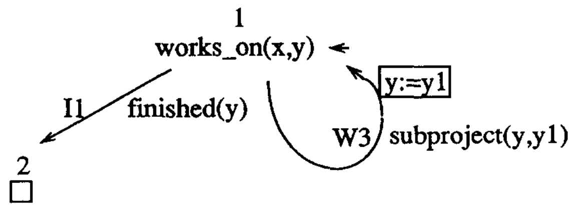

Fig. 10 Compilation graph for the works-on problem.

works-on(x, y)←. We fold on it to obtain the compilation graph of Figure 10.

With the meta-predicate $p/1$ to represent the loop-node, we obtain the clauses:

$N1$: p(works-on(x, y)) ← finished(y). (state 1 to state 2)
$N2$: p(works-on(x, y)) ← subproject(y, $y1$), p(works-on(x, $y1$)).
(state 1 to state 1)

Several comments need to be made regarding this application. First, it is not sensible to compile an integrity-checking program if updates of the given type do not occur frequently. Thus, in general, we do not compile a program that checks for possible integrity violations caused by inserting or deleting a rule or an integrity constraint. As an example, consider updates such as insertions of the type parent(x, y)←, with x and y ground. They may occur frequently in a database. On the other hand, the insertion of a new rule, such as father(x, t) ← parent(x, y), male(x), will only occur once.

A second comment in the same spirit is that whenever the integrity checking involves a call to a predicate which is frequently updated, then this call should not be expanded in the trace, but represented – and solved – as a label on an arc (e.g., finished/1 and subproject/2 in the example). By doing so, we do not have to recompile the program after each update for this predicate. In fact, in our example, a recompilation is only needed whenever a rule or an integrity constraint which includes a call to works-on/2 in its body is inserted or deleted.

In [6] recursive rules are not explicitly considered. However, the application of the compilation method described in that paper to our simple example, produces (among other ones) essentially the same clauses. The main advantage of our method, is that the clauses we synthesize are a precise compilation of the control which was expressed by a user (or oracle). In more complication situations than described in the example, sophisticated additional control might be necessary. In such circumstances, our method provides a way to impose this control. (See also the remark below.) In general, the method of [6] produces many more clauses, independent of any underlying control strategy.

Finally, there is the problem of finding an efficient search-strategy. In the presence of recursive rules and using a mixed computation strategy, integrity checking can easily produce infinite loops. General principles for obtaining useful strategies have been proposed in [18–20]. In our example, we have followed [18], by continuously generating new facts derivable from the update and then consuming them to check whether the integrity constraints have not been violated. Furthermore, in order to avoid computing the same fact a number of times, we can keep a record of the information that has been derived so far. Finally, we can safely introduce a 'cut' at the end of every clause synthesized by the compilation procedure, since one refutation is sufficient to establish the fact that the integrity is violated. If we assume that in our example a user will specify his intended update with a query $\leftarrow$ add(works-on(A, B)), then these observations result in the following clauses:

$A1$: add(works-on(x, y)) $\leftarrow$ assert(addition(works-on(x, y))), p(works-on(x, y)), !, write("integrity violated"), retractall(addition($-$)).
$A2$: add(works-on(x, y)) $\leftarrow$ retractall(addition($-$)), assert(works-on(x, y)).
$P1$: p(works-on(x, y)) $\leftarrow$ finished(y),!.
$P2$: p(works-on(x, y)) $\leftarrow$ subproject(y, z), not(addition(works-on(x, z))), assert(addition(works-on(x, z))), p(works-on(x, z)).

Here, we have assumed that a user will specify his intended update with a query $\leftarrow$ add(works-on(A, B)). Clearly, in a database environment, these clauses should not be executed using the tuple-at-a-time mechanism of PROLOG.

3.3. THEOREM PROVING

With the mixed computation rule of the previous example, we already introduced an element of theorem proving. However, the program contained only Horn clauses. In our next example, we illustrate that we can just as well start off with a logical problem specification in full (non-Horn) clausal form, apply a theorem proving strategy to it and compile the computation into a top-down executable (PROLOG) program.

The example is inspired by Moore's three blocks problem. Given is a pile of blocks of infinite height. Every even block is red and at least one out of every three connecting blocks is blue. The problem is to find the blue blocks. In clausal form, we have

$C1$: col(0, Red)$\leftarrow$.
$C2$: col(n, Red) $\leftarrow$ $n > 1$, plus($n2$, 2, n), col($n2$, Red).
$C3$: col(n, Blue), col($n1$, Blue), col($n2$, Blue) $\leftarrow$ plus($n1$, 1, n), plus($n2$, 1, $n1$).
$C4$: $\leftarrow$col(n, Red), col(n, Blue).

The desired mixed control rule for the query $\leftarrow$ col(m, Blue) is expressed in the trace fragment of Figure 11.

The reorganized graph is drawn in Figure 12. The leaf, node 7, is a renaming of the previous node 5. It is the only loop-node. By folding on it, we obtain the compilation graph. Its shape should be clear from the previous examples, so we omit it.

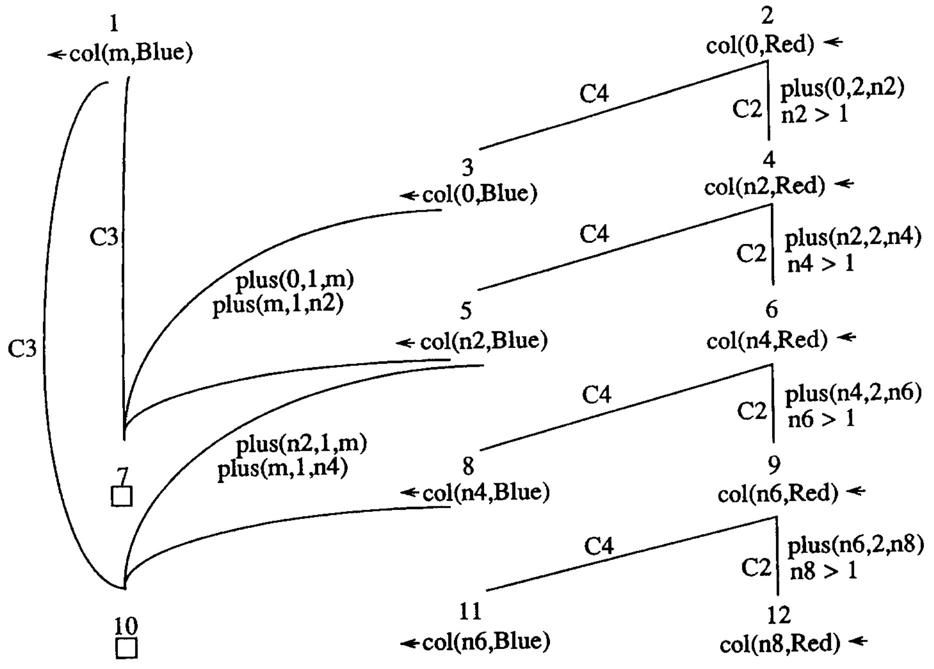

Fig. 11. Fragment of the trace for the pile-of-blocks problem.

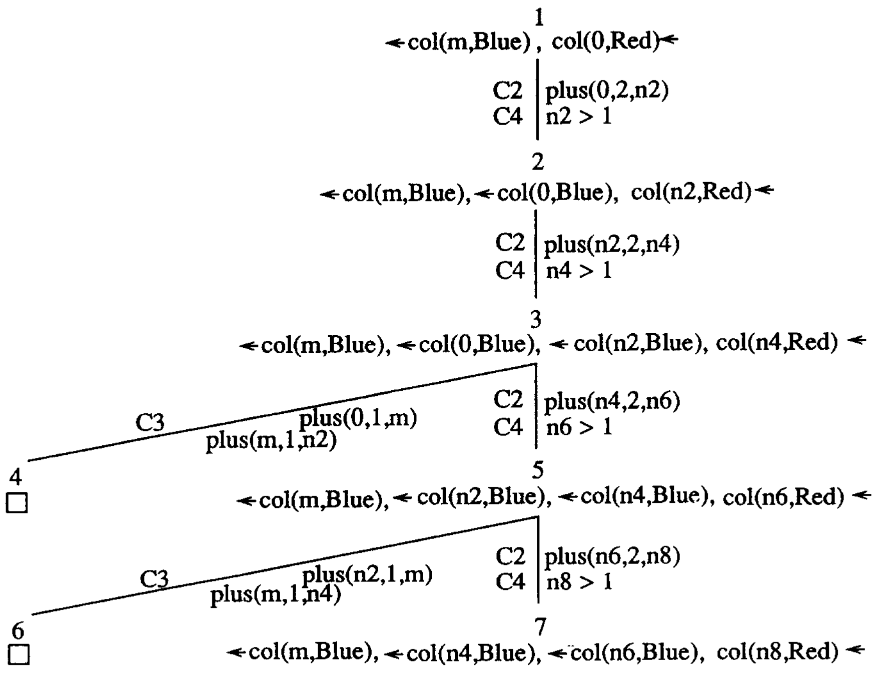
1
←col(m,Blue), col(0,Red)←
C2
C4
plus(0,2,n2)
n2 > 1
2
←col(m,Blue), ←col(0,Blue), col(n2,Red)←
C2
C4
plus(n2,2,n4)
n4 > 1
3
←col(m,Blue), ←col(0,Blue), ← col(n2,Blue), col(n4,Red) ←
C3
plus(0,1,m)
plus(m,1,n2)
C2
C4
plus(n4,2,n6)
n6 > 1
4
5
←col(m,Blue), ← col(n2,Blue), ←col(n4,Blue), col(n6,Red) ←
C3
plus(n2,1,m)
plus(m,1,n4)
C2
C4
plus(n6,2,n8)
n8 > 1
6
7
←col(m,Blue), ←col(n4,Blue), ←col(n6,Blue), col(n8,Red) ←

Fig. 12. Reorganized trace tree for the pile-of-blocks problem

By adding a new root node, (0, $\leftarrow$col(m, Blue) – as in the Fibonacci example – and with the meta-predicate $p/4$ representing the loop-node, we obtain the (Horn clauses):

$N1$: col(m, Blue) $\leftarrow$ plus(0, 2, $n2$), $n2 > 1$, plus($n2$, 2, $n4$), $n4 > 1$, plus(0, 1, m), plus(m, 1, $n2$). (state 0 to state 4)

$N2$: col(m, Blue) $\leftarrow$ plus(0, 2, $n2$), $n2 > 1$, plus($n2$, 2, $n4$), $n4 > 1$, plus($n4$, 2, $n6$), $n6 > 1$, p(col(m, Blue), col($n2$, Blue), col($n4$, Blue), col($n6$, Red)). (state 0 to state 5)

$N3$: p(col(m, Blue), col($n2$, Blue), col($n4$, Blue), col($n6$, Red)) $\leftarrow$ plus($n2$, 1, m), plus(m, 1, $n4$). (state 5 to state 6)

$N4$: p(col(m, Blue), col($n2$, Blue), col($n4$, Blue), col($n6$, Red)) $\leftarrow$ plus($n6$, 2, $n8$), $n8 > 1$, p(col(m, Blue), col($n4$, Blue), col($n6$, Blue), col($n8$, Red)). (state 5 to state 5)

As in the previous examples, the calls to $>/2$ and the functors (and constructs) in $p/3$ are redundant. However, the logic program is an efficiently executable compilation of the desired proof-strategy.

Again, the problem of how to obtain an efficient strategy for solving the problem at hand is non-trivial. We do not claim any contribution to this issue, and we refer to [21] for a good overview of the achievements in the field. The benefit of our technique is that we can use a general theorem prover to detect a good inference strategy for a given problem and then compile the successful derivation paths into efficiently executable programs. Clearly, this poses a problem of completeness. Therefore, the application domain we are mostly aiming at is that of machine learning. In this field, one of the important issues is to find techniques which can derive efficient new rules for relevant problems concerning a given knowledge base. This is precisely what the combination of a general-purpose theorem prover and our compilation technique will achieve for this type of problems.

4. Discussion

Our main achievement in this paper is that we have shown that the transformation technique called 'Compiling Control', which was designed to compile coroutining, can also be used to compile bottom-up and mixed computations. We have illustrated how this can be applied to solve several state-of-the-art problems in different domains: eliminating redundant computations, transformation almost-tail-recursive procedures into tail-recursive ones, compiling integrity checking and learning new, efficiently executable rules to solve given problems concerning a problem domain (expressed in (non-Horn) clausal logic).

We deliberately started off from the very restricted Definition 2.2 for the compilability of triplets (P, q, r). Most of the examples studied in [1 and 11] do not satisfy this condition. In these examples, there does not exist a finite fragment F of the computation trace, such that every leaf of F is a renaming of a previous state. Also, the notion of *consistency* in [11] is a more general than that of Definition 2.1. Still, these more general triplets can be compiled. The reason why we restricted our attention to the subset of triplets (P, q, r) of Definition 2.2 (and Definition 2.5) is that we wanted to focus on the new applications of the technique. Generalizing the entire framework of [11] to cover non-top-down computations would have forced us to go into many more technical details. Also, such an approach – in addition to presenting all the applications – would not have been possible within reasonable space-constraints.

A typical example of how this decision limits the general applicability of the technique (as it was presented here), is the pile-of-blocks problem. If we choose to represent the integers in the first argument of col/2 with a successor-function, $s/1$, then, with the same control rule r as in 3.3, we obtain a triplet (P', q', r) which is not compilable. The reason is that the (new representations for the) states in node 5:

$$\leftarrow \mathrm{col}(n, \mathrm{Blue}), \leftarrow \mathrm{col}(s(s(0)), \mathrm{Blue}), \leftarrow \mathrm{col}(s(s(s(s(0)))), \mathrm{Blue}),$$
$$\mathrm{col}(s(s(s(s(s(s(0)))))), \mathrm{Red}) \leftarrow$$

and node 7:

$$\leftarrow \mathrm{col}(n, \mathrm{Blue}), \leftarrow \mathrm{col}(s(s(s(s(0)))), \mathrm{Blue}), \leftarrow \mathrm{col}(s(s(s(s(s(s(0)))))), \mathrm{Blue}),$$
$$\mathrm{col}(s(s(s(s(s(s(s(s(0)))))))), \mathrm{Red}) \leftarrow$$

are no longer renamings. In fact, no two states will ever be renamings. However, it is clear that both states are instances of a same more general pattern:

$$\mathrm{col}(n, \mathrm{blue}), \leftarrow \mathrm{col}(s(s(x)), \mathrm{Blue}), \leftarrow \mathrm{col}(s(s(s(s(x)))), \mathrm{Blue}),$$
$$\mathrm{col}(s(s(s(s(s(s(x)))))), \mathrm{Red}) \leftarrow .$$

A more general framework for the compilability of control rules can be sketched as follows; let (P, q, r) be a triplet consisting of a program, query, and control rule and T a fixed, finite set of states (again, conjunctions of clauses), such that

- r is a consistent control rule, in the sense that for every state S, which is an instance of a state S', with S' in T, the actions prescribed by r in S are a corresponding instance of the set of fixed actions prescribed by r in S'. Here, we allow exceptions for these states S that belong to a well-identified initial part of the computation trace, describing an initialization phase,
- starting from any general pattern, $S1$ in T, and performing the actions prescribed by r, we obtain, after a finite number of inference steps, an instance $S2'$ of a state $S2$ in T.
- there exists a finite fragment F of the computation trace (strictly exceeding the initialization phase) such that every leaf S of F is an instance of a state S' in T,

then (P, q, r) is compilable.

References

1. Bruynooghe, M., De Schreye, D., and Krekels, B., 'Compiling control', *J. Logic Programming*, **6** 135–162 (1989).
2. Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, MIT Press, Cambridge, 1989.
3. De Schreye, D. and Bruynooghe, M., 'The compilation of forward checking regimes through meta-interpretion and transformation', in *Meta-programming in Logic Programming*, H. Abramson and M. H. Rogers (Eds) MIT Press, 1989, pp. 217–231.
4. Bruynooghe, M., De Raedt, L., and De Schreye, D., 'Explanation based program transformation', in *Proc. IJCAI'89*, 1989, Detroit, pp. 407–412.
5. Debray, S K., 'Unfold/fold transformations and loop optimization of logic programs', in *Proc. SIGPLAN'88* Conf. on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 23, No. 7, July 1988, pp. 297–307.
6. Kowalski, R., Sadri, F., and Soper, P., 'Integrity checking in deductive databases', in *Proc. 13th VLDB*, Brighton, England, 1987.
7. Sadri, F. and Kowalski, R., 'A theorem-proving approach to database integrity', in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Morgan Kaufmann Publishers, 1988, pp. 313–362.
8. Soper, P. J. R., *Integrity Checking in Deductive Databases*, M.Sc. Thesis, Department of Computing, Imperial College, London, 1986.
9. Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T., 'Explanation-based generalization: a unifying view', *Machine Learning*, **1**, 47–80 (1986).
10. DeJong, G. and Mooney, R., 'Explanation-based learning: an alternative view', *Machine Learning*, **1**, 145–176 (1986).
11. De Schreye, D. and Bruynooghe, M., 'On the transformation of logic programs with instantiation based computation rules', *J. Symbolic Computation*, **7**, 125–154 (1989).
12. Debray, S. K. and Warren, D. S., 'Detection and optimization of functional computations' in *Prolog, Proceedings Third International Logic Programming Conference*, LNCS, 225, Springer-Verlag, 1988, pp. 490–504.
13. Kanamori, T., Horiuchi, K , and Kawamura, T., 'Detecting functionality of logic programs based on abstract hybrid interpretation', *ICOT Technical Report*, TR 331, 1988.
14. Clocksin, W. F., 'A technique for translating clausal specifications of numerical methods into efficient programs', *J. Logic Programming*, **5** (3), 231–242 (1988).
15. Burstall, R M. and Darlington, J. 'A transformation system for developing recursive programs', *JACM*, **24**, 44–67 (1977).
16. Debray, S. K., *Global Optimization of Logic Programs*, Ph.D. dissertation, Stony Brook, 1986.
17. Bruynooghe, M. and De Schreye, D., 'Some thoughts on the role of examples in program transformation and its relevance for explanation based learning', in *Proc. International Workshop on Analogical and Inductive Inference*, (AII89), LNCS, 397, Springer-Verlag, 1989, pp. 60–78.
18. Decker, H. 'Integrity enforcement on deductive databases', in *Proc. First International Conference on Expert Database Systems*, Charleston, South Carolina, USA, 1986.
19. Lloyd, J. W., Sonenberg, E. A., and Topor, R. W., 'Integrity checking in stratified databases', *J. Logic Programming*, **4**, 331–343 (1987).
20. Bry, F., Decker, H., and Manthey, R., 'A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases', in *Proc. EDBT'88*, Venice, Italy, 1988.
21. Ramsay, A., *Formal Methods in Artificial Intelligence*, Cambridge University Press, 1988.