Advanced Topics in Automated Deduction

by
Wolfgang Bibel

Technical Report 87-39

November 1987

Advanced Topics in Automated Deduction*

W. Bibel

University of British Columbia and
Canadian Institute for Advanced Research

1 Introduction

This chapter deals with a number of issues in Automated Deduction that have recently attracted some attention in this area. The presentation is *not* meant to provide an introduction to this area. Rather its purpose is to supplement the introductory articles [5,24] contained in a preceding volume. In other words, the reader is expected to be familiar with the basic techniques used in Automated Deduction either from these or from other sources. This requires, for instance, a familiarity with resolution and the connection method. Also, the importance of deduction for Artificial Intelligence is not discussed here again. The reader who lacks motivation (or is confused by some ongoing discussions) in this respect might wish to read the introduction to [5].

In recent years there has been a remarkable progress in the field of automating deduction which is mainly due to the success of PROLOG. The number of logical inferences per seconds (LIPS) now ranges between 300K and 500K for the most advanced systems. Ironically, this dramatic speed–up in deductive performance has not yet produced any significant result such as the proof of a longstanding conjecture in Mathematics, or the like. Rather it occasionally happened that seemingly simple deductive problems cause these systems difficulty. This, for instance, happened with Schubert's steamroller problem [12] that we will briefly discuss in Section 3.

We see the following three different main problems in the current technology of Automated Deduction.

- Control regimes such as those used in PROLOG are too simple–minded.

*To appear in "Fundamentals of Artificial Intelligence II", R. Nossum, ed., Springer (LNCS), Berlin

- The expressive power of PROLOG is too poor.
- Certain phenomena arising in human reasoning are badly understood for an adequate representation within some logical formalism.

In this chapter we will not address the last item, only briefly discuss issues related with the second item, and focus mainly on the first of these points. Indeed, a more elaborate control regime for deductive systems may enhance their performance without affecting the virtues of the architecture of PROLOG–like systems as has been demonstrated in [12]. In order to establish a basis for the discussions of this issue, we begin our presentation with studying various possible control regimes. First, we review the standard arguments in Section 2 that favor top–down in comparison with bottom–up control. Then we show in Section 3 that a reasonably restricted bottom–up regime is actually superior in its performance. The gist of this regime consists of the application of several reduction operations, notably what we call DB– and ISOL–reductions. Only after their application is the control handed over to a top–down regime. One may think of this approach as a way of eliminating some of the redundancy involved in the representation of a logic program in order to then work on the harder core of it.

Section 4 extends these discussions to programs that contain recursive rules. Such rules give rise to deductive cycles. So Section 4.1 first characterizes such recursive cycles and distinguishes them from tautological ones. Section 4.2 then describes an algorithm that compiles a certain class of programs into iterative mechanisms. The application of this algorithm is then demonstrated with more complicated problems, namely the Fibonacci program and the rule expressing the associative law that both illustrate specific features. Factoring plays a significant role for the first and a resulting organization of the database does so for the second of these problems. Since there is a great similarity with the work done in datalogic, but also some terminological confusion in the database literature, we include a clarification of some notions in Section 4.5. Finally, the limitations of our approach are discussed.

Note that the approach we take is quite different in its spirit from that aiming at LIPS performance records. An elaborate analysis like the one required for the algorithm just mentioned needs quite a bit of execution time, which might slow down the performance for simple problems. We gain a more intelligent behavior this way, however, so that in more elaborate problems the advantages will outweigh the overhead.

Although quantifiers are present in most of human reasoning, current deductive systems of the PROLOG–type do not even mention them. In Section 5 we recall a way dealing with them that has long been neglected. It does not

take recourse to skolemization, but rather takes advantage of the logical operators in their relative positions within the formula. While this is already attractive in the case of first–order logic, it has recently led to an extremely elegant solution to treating deduction in various modal logics [26] that we briefly summarize. A brief section reminding of the importance of building–in theories and mathematical induction concludes the whole chapter.

In our treatment we use standard notation as much as possible. In cases of doubt the reader might wish to check with [3].

2 Bottom–up vs top–down execution

The control regime of a PROLOG system is strictly top–down. It takes the goal clause (considered to be at the top of a search tree), selects its first literal, searches for a matching head, performs resolution on the pair of literals (i.e. on the connection) thus located, yielding a modified goal clause for which this process is carried out in exactly the same way again, and so forth. This regime, which is actually based on *ordered input resolution*, works fine for many problems. At the same time for many other problems it is just not appropriate and often fails badly. Consider first why it works pretty well in many cases, which will be illustrated by the following two programs.

$$
\begin{array}{lll}
Pa_1. & & \\
\vdots & & \\
Pa_m. & & \\
Q_1x & \leftarrow & Px. \\
 & \vdots & \\
Q_nx & \leftarrow & Px. \\
 & \leftarrow & Q_na_m?
\end{array}
$$

Top–down this program would succeed with two inference steps in the obvious way. An unintelligent bottom–up execution would need $m \cdot n$ inference steps, not counting in both cases the identification of pairing predicates. As we can see, top–down execution in PROLOG outperforms bottom–up execution in this kind of program.

$$
\begin{array}{lll}
P0. & & \\
Q0. & & \\
P(x+1) & \leftarrow & Px. \\
Q(x+1) & \leftarrow & Qx. \\
 & \leftarrow & Qn?
\end{array}
$$

Top–down this program would succeed with $n+1$ steps (see below for the built–in arithmetic) while, again, an unintelligent bottom–up execution might actually run forever. These two programs seem to be typical for programs involving a number of facts and "chaining" rules on the one hand and recursive rules on the other. Since facts, chaining and recursive rules are the main components of any logic program, it seems that in general top–down execution provides a much better performance. This is not always true as we will see upon closer inspection in the following sections.

Let us start considering this other side of the coin with a look at the factorial program below which is actually a variant of the recursive program above.

$$\begin{array}{lll} FAC(0,1)\,. & & \\ FAC(x+1,(x+1)\cdot y) & \leftarrow & FAC(x,y)\,. \\ & \leftarrow & FAC(n,z)\,? \end{array}$$

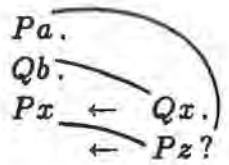
In the discussion of this and similar examples we allow theory connections and theory unifications [3], i.e. terms like $0+1$ and $1\cdot 1$ are considered unifiable on the basis of the theory of arithmetic. In this sense this theory is *built into* the deductive mechanism. Now, if this program is executed top–down, any of the instances of the variable y cannot be assigned a value until the whole process reaches the bottom which requires the intermediate storage of n expressions associated with these n instances that cannot be evaluated. In contrast, the bottom–up execution allows the immediate computation of the value of any instance of each occurring variable. Since this value is all that is needed in order to compute the value of the next instance, this next instance may make use of the same storage location as the previous one, which altogether amounts to three storage locations for the three occurring variables. Even without a more detailed quantitative analysis it should be obvious from this discussion that here bottom–up execution results in a much faster computation of the final value for z than top–down execution.

In [9] (see also [7]) it was first shown that bottom–up execution corresponds to the execution of iterative programs in conventional programming languages, and top–down to recursive ones. The implementation of recursion in general requires maintaining a data structure (called a stack–frame) for every recursive call that has not terminated yet. A recursive computation involving n recursive procedure calls requires, therefore, space linear in n. On the other hand, an iterative program typically uses only a constant amount of memory, independent of the number of iterations. These observations apply for any programming language, and have been illustrated for PROLOG above with the factorial program.

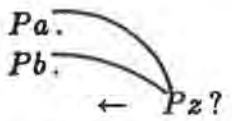
This demonstrates that there seem to be virtues in bottom–up execution as well. This is true not only for recursive programs but also for programs with a focus on facts and chains. An example is Schubert's steamroller problem that will be described in the following section. So the natural question arises whether one might combine the virtues of both kinds of regimes. We devote the following sections to a discussion of exactly this question. This discussion will result in a much more attractive control regime than the one provided by either top–down or bottom–up execution.

3 Bottom–up reductions

As we know from [24] (or from [3] for that matter) the problem of finding a proof consists of identifying an appropriate subset among the connections of the given formula. Obviously, the smaller the set of connections the easier this task can be achieved. There are many possibilities to reduce this set in a given formula. A number of them are treated, for instance, in [3] such as pure literal reduction, subsumption, tautology reduction, and elimination of simple circuits, that all remove connections and possibly clauses. Here we concentrate on further reductions of a less familiar sort. Let us consider an example for illustration.



Concentrating on the Q–connection we observe that both literals engaged occur in no other connection of the whole formula. We call connections with this property *isolated*. Note that the remaining connections are not isolated in this sense, since Pz occurs in both of them. The isolated connection in question may be eliminated by substituting the affected clauses by their resolvent, which amounts to a (bottom–up) resolution step with elimination of the parent clauses and has the following result.



Clearly, this reduction step, called *ISOL–reduction*, decreases the complexity of the initial formula and retains provability in general.

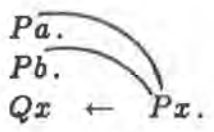
It can easily be seen that the definition of isolated connections may be liberalized by allowing literals to occur in more than one connection if they are

ground literals in unit clauses (i.e. facts). For details on this and for several ways to produce isolated connections see [12]. Here we only discuss one further reduction that is based on the following valid equivalence.

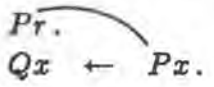
$$Pr \leftarrow r \in \{c_1, \ldots, c_k\} \;\leftrightarrow\; Pc_1 \wedge \ldots \wedge Pc_k$$

The substitution in a logical formula of the right side of this equivalence by the left side will be called a *DB–reduction*. Note that the literal $r \in \{c_1, \ldots, c_k\}$ may be tested for its truth–value by database operations. For that reason we will take the convention to drop such a literal from this kind of logic program and treat the range that it defines separately (or even implicitly). In fact we take the view that this range is actually represented in a database table not presented explicitly in the following examples.

DB–reduction properly reduces the size of a formula. But more importantly, it often results in new isolated connections which may then in turn be eliminated. For instance,

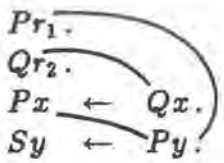


has no isolated connection, while after DB–reduction, yielding

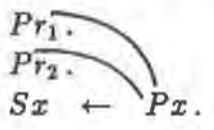


with $r \in \{a, b\}$, indeed an isolated connection appears.

In conjunction with ISOL–reduction DB–reduction in general requires the use of all the usual database operators such as union $\cup$, intersection $\cap$, projection π, and the (natural equi–) join $\bowtie$. Since they are standard [25], we only demonstrate them by use of a few simple examples. For instance, consider the following program in which $r_1 \in R_1$ and $r_2 \in R_2$.

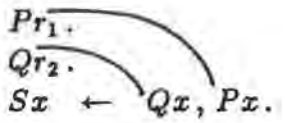


Here only the Q–connection is isolated which upon removal yields



The obvious next step is a further DB–reduction that takes the union $R_1 \cup R_2 = R$ for the range of r.

obviously producing a new isolated connection. It is well–known in logic programming that two subgoals in a clause sharing common variables correspond to the join in databases which in the simplest case of unary predicates is identical with the intersection $R_1 \cap R_2 = R$. For instance, if r ranges over R thus defined then ISOL–reduction upon the two connections in the program



simply results in Sr. Had the subgoals been Qyx, Pxz, we would have had $R_1 \bowtie R_2 = R$ instead with everything else remaining the same. The projection, finally, comes into play if Pr is connected with a clause of the form $Qx \leftarrow Pxy$. ISOL–reduction here leads to Qr_1, where $r_1 \in \pi_1 R$.

With these reductions at hand, an enhanced control regime would now operate on a given logic program in the following way. All the possible connections will first be determined. Then possible reductions will be performed until none is applicable any more. To some extent these operations may already be performed even at compile time. Note that they never require more than linear time (otherwise they should not be subsumed under the notion of a reduction, cf. [6]). Thereafter, the usual top–down execution will be carried out on the remaining program. Note that in this last phase a form of unification is needed that incorporates database operations.

As a technical remark we mention that in performing ISOL–reduction with a literal Pr in a unit clause this literal would not actually be removed from the program. As a reference to a database table it might be needed for further queries to the system that might involve the predicate P.

A proof system for Horn clause logic, PROTHEO, that follows this whole scheme of operation has been developed in the author's former research group [12]. In [12] the power of this approach has been demonstrated with Schubert's steamroller problem. This problem describes the eating habits of a number of animals, and asks for the existence of an animal among these characterized by certain properties. As a logical formula it consists of 26 clauses (in one of the possible formalizations). For most proof systems operating in a purely top–down oriented way the search space of this formula turned out to be infeasible.

DB– and ISOL–reduction reduces these 26 clauses to merely 4 remaining clauses with altogether 7 literals. There is nearly no search left so that the solution can be computed in a straightforward way mainly as a deterministic sequence of fast database operations.

A number of other proposals have been made to cope with the problem that was demonstrated by the steamroller formula. Most if not all of them involve the declaration of sorts, requiring a sorted logic theorem prover. In effect this eventually results in a behavior very similar to the one in our solution. This is because a DB–reduction actually collects several items into the same "sort". By ISOL–reduction these sorts are then propagated through the remaining formula. In sorted logic we have a more elaborate unification procedure that additionally has to care for the sorts of the terms. In our solution the analogue complication is the incorporation of database operations into the unification mechanism mentioned above.

While no one has studied these similarities and potential differences in further detail, it seems to be obvious that technically the differences are negligible. However there is a major drawback for the sorted logic approach insofar as it requires the programmer to provide the system with the information as to which among the unary predicates are to be treated as sorts, an unpleasant extra burden on the user. Also it does not seem to provide a solution for "sort"–predicates with more than one argument, while obviously our reduction mechanism is completely general in that respect. Since first–order logic is already complicated enough, particularly in view of more general applications like in non–monotonic reasoning, the extra technical complications involved in sorted logic cannot be regarded as a particular virtue either. Hence for all these and other reasons the reduction mechanism described in this section is our favorite solution to this issue.

By now the reader may have noticed that the first program in the previous section, which was meant to witness the supposed superiority of top–down vs. bottom–up execution, with DB– and ISOL–reduction (i.e. bottom–up) behaves as efficiently as the usual top–down execution. In other words the arguments given there — for the case of facts and chaining rules — have lost their credibility. On the contrary, a bottom–up regime like the one we just described is at least as good as a purely top–down one, and sometimes (like in the steamroller problem) even dramatically better. It replaces deep search trees by shallow ones, thus reducing the need for costly backtracking substantially.

In [4] it is shown that the same technique leads to the reduction of any kind of a constraint satisfaction problem to the problem of executing a nested term consisting of database operations. Since constraint satisfaction problems play

a particularly important role in Intellectics[1], we see that the seemingly simple ideas presented in this section apparently have important applications beyond those represented by the steamroller problem.

In spirit the Q^* algorithm [19] addresses exactly the same problem as the one dealt with in the present section. The authors actually emphasize their view of Q^* as a problem reduction mechanism which is exactly what our reduction rules provide as well. While here we presented a very few and simple reduction rules, Q^* seems to be a collection of a number of ideas built into this algorithm. As far as it can be taken from the way Q^* is presented in [19] all these ideas are actually covered in a technical sense by our reductions, so that our approach appears to be much more satisfactory because of its simplicity, generality, and uniformity. But it might still be worthwhile to look into the (unpublished) algorithm Q^* itself in order to possibly detect features that might further enhance our technique perhaps by way of a further reduction rule.

4 Recursion

So far we have discussed a number of reduction operations that simplify a given logical problem and sometimes already provide even the whole proof. We now study the logical structure of the formulas that result from such a sequence of reductions, aiming at the identification of further possibilities for their efficient treatment. Such formulas will typically contain cycles of connections possibly indicating some sort of a recursive structure. The present section will hence be devoted to the topic of recursion.

4.1 Cycles

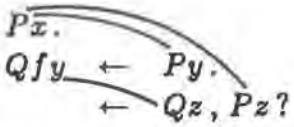
We begin our discussion of recursion with the definition of the more general concept of a cycle.

Definition. In a set of clauses a *cycle* is a set $\{c_1, \ldots, c_n\}$ of connections $c_i = \{L_i, R_i\}$, $i = 1, \ldots, n$, such that R_i and L_j with $j - i = 1 \bmod n$ are different literals in one clause. A cycle is called *linear* if there is no literal that occurs in more than one of its connections.

Recall that unifiability of the terms in the literals is not inherent in the notion of a connection, that is, only the predicate symbols determine connections. Our notation $\{L_i, R_i\}$ is meant to indicate an implicit existential quantification that orders the otherwise unordered pairs of literals in the connections. The following example illustrates a cyclic set of connections that would not qualify as

[1] the field of Artificial Intelligence and Cognitive Science

a cycle in the sense of our definition, since Px is contained in two connections next to each other.



The cycle formed by the two connections in $Axz \leftarrow Axy, Ayz$ is not linear since Axz occurs in both of them. There are two different types of such cycles, the tautological and the recursive ones. Let us first consider tautological cycles.

Definition. A cycle is called *tautological* if there exists a clause C involved in the cycle and a substitution which on C is the identity substitution, such that all connections in the cycle become complementary pairs of literals.

For instance, the cycle in each of the following clause sets is tautological,

$$\{\{\overbrace{Px \leftarrow Qy, Px}\}\}, \quad \{\{\overbrace{Px \leftarrow Qy\}, \{Qz \leftarrow Pz}\}\},$$

$$\{\{\overbrace{Pxy \leftarrow Qyx\}, \{Qza \leftarrow Pza}\}\}$$

while this does not hold for the following clause sets.

$$\{\{\overbrace{Px \leftarrow Py}\}\}, \quad \{\{\overbrace{Axz \leftarrow Axy, Ayz}\}\},$$

$$\{\{\overbrace{Px \leftarrow Qx\}, \{Qa \leftarrow Pb}\}\}, \quad \{\{\overbrace{Pxy \leftarrow Qxy\}, \{Qza \leftarrow Pua}\}\}$$

In no case can tautological cycles ever give rise to a recursion since the identity substitution on C in the definition prevents any progress while traversing the cycle in a deductive process. Under certain conditions even some or all of the connections in a tautological cycle may be deleted without affecting provability. On the ground level such conditions have been stated in Lemma 3.6 of [6] for a class of tautological cycles called simple circuits (mentioned before in section 3). The generalization to the first-order level follows standard techniques [3], which always require some care, however. The deletion of such connections may leave some of the previously connected literals pure in the sense of [6] so that whole clauses may become obsolete. Altogether this is a powerful generalization of the well–known reduction of deleting tautologies mentioned already in the previous section. It may be used to further reduce a given program beyond what was already achieved with the techniques described in the previous section, and in this sense it is indispensable for the restriction of the search space.

Definition. A cycle is called *potentially recursive*, if it is not tautological and if there exists a weak unifier for all its connections.

Weak unification was defined in [15] and captures whether in a proof a pair of literals is potentially unifiable by possibly renaming variables so that a

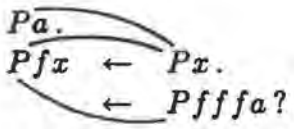
variable occurring at both ends of a connection adopts a name at one end that is different from the one at the other. For instance, the connection in

$$\overbrace{Pfx \leftarrow Px}.$$

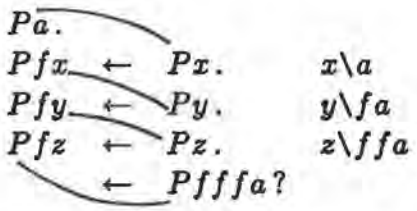
is weakly unifiable in this sense since x will be renamed to, say, x' in one of the two literals. The obvious idea behind this concept is to take into account the possibility of considering more than a single copy of the clauses involved.

Definition. A cycle is called *recursive* if there is a proof of the formula with a spanning set of connections that contains a fixed number of instances of the whole cycle, i.e. of all its connections.

For instance, there are two such instances (depicted here with one connection only) from the potentially recursive cycle of the previous example in the following proof.



If the instances of the clauses involved in a recursive cycle are represented explicitly, then there is actually no cycle anymore. This may be seen in the following picture where, along with the substitutions involved, the three instances of the rule in the previous proof here are represented explicitly.



Nevertheless there is this cyclic property of coming back again and again to a copy of the same clause. In the following we will often briefly speak of a recursive cycle even if we actually mean a potentially recursive one. This notion of a recursive cycle, usually introduced with resolution in mind, is generally accepted although there are differences in the details of the definition given by various authors such as our exclusion of tautological cycles; this will be further discussed in Section 4.5 below.

The restriction in our definition of cycles, whereby the literals in the same clause have to be different ones, excludes other types of cycles (like the example shown above) that might as well be considered in our context. Although I lack a striking argument I nevertheless feel that the notion captured here is the

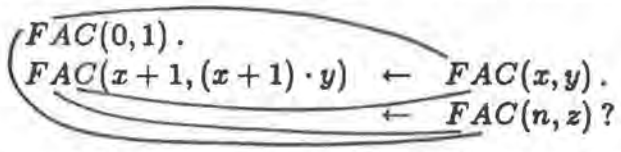
more productive one for our purposes. Indications of this are the following two observations. First, a tour through a cycle, that ends in the same literal where it started from, indeed returns to this point of departure in quite a different state than it started from since, for instance, it could not run through the same cycle again in a deductively meaningful way. Second, these types of cycles seem better attacked with the reduction provided by factoring, which reduces them to the cycles we have defined above, or the cycle can at least be broken into two parts associated with the deductive solution of two independent subgoals. Both actually applies to the example of a "non–cycle" given above.

Recursive cycles as defined above may still include cycles that are tautological in nature (but of course not tautological in the sense of our definition). We may only conjecture that our definition of tautological cycles covers all the cycles that are indeed tautological in nature. It is actually somewhat of a surprise that no one seems to have bothered before to clarify such a basic distinction.

4.2 Compilation of recursive cycles

In a given formula (or logic program) all cycles can be detected fast by well–known algorithms. Once the cyclic structure is known, this insight may be used for a much faster proof detection than the one that could be achieved by a straightforward depth–first search as in PROLOG. This observation was known for a long time and has been made, for instance, in [9]. Special attention was given recently to this possibility by the researchers working in datalogic, i.e. in the field on the boundary between logic and databases that will be reviewed in more detail in Section 4.5 below.

The most attractive approach among those taken in this kind of work is the one that aims at a compilation of the whole proof process [17]. For instance, the compilation of a program such as the one in our last example in the previous section should result in a code that checks the term in the goal for an arbitrary number of consecutive f's followed by an a, rather than performing any deductive steps at all [9]. We will now outline such an approach for a limited class of programs. First, let us illustrate the idea with the familiar factorial problem.



We have discussed this program already in Section 2. Recall from there that we allow theory connections and theory unifications [3], i.e. terms like $0+1$ and $1\cdot 1$ are considered unifiable on the basis of the theory of arithmetic. There are four unifiable connections in this program and one of them forms a

potentially recursive cycle. How could one, with such a knowledge at hand, head for a compilation such that any reasonable goal may be executed fast without explicit (more time-consuming) deduction steps?

Recall that running a logic program means finding and executing a proof. Also recall that in terms of the connection method a proof consists of a spanning set of connections. The cycle does not itself form a spanning set of connections. The single connection formed by the fact $FAC(0,1)$ and the top goal would satisfy this requirement, but in this simple case the cycle connection would not even be involved in the proof. Another spanning set of connections is set up by the connection formed by the fact $FAC(0,1)$ and the rule subgoal, the cycle connection, and the connection formed by the rule head and the top goal. Let us examine more closely this second alternative.

Since the cycle connection is not tautological, it must connect two different instances of the rule which we may assume to be the i–th and the $i+1$–th instances. The unifications that are implied by this connection may be expressed as the following equations.

$$x_{i+1} = x_i + 1\,,\ \ y_{i+1} = (x_i + 1) \cdot y_i$$

The $FAC(0,1)$–connection may be used to determine a value for the base case.

$$x_1 = 0\,,\ \ y_1 = 1$$

The top goal connection determines the value for the output.

$$z = (x_k + 1) \cdot y_k$$

where k is such that $x_k + 1 = n$. It is a simple step from these equations to the following destructive assignment program.

$$x \leftarrow 0;\ \ y \leftarrow 1;\ \ \ell : x \leftarrow x + 1;\ \ y \leftarrow x \cdot y;\ \ \text{if } x \neq n \text{ goto } \ell;\ \ \text{output } y$$

This program clearly outperforms any deductive mechanism, or, to put it differently, it provides the fastest way to carry out the proof for the above formula (except for parallel executions). So is there a mechanical way to extract it from the formula? We claim that the following *compilation algorithm* may be refined so as to achieve this for a limited class of formulas.

1. Identify cycles in the problem (the single connection in the rule above).
2. Select one of the cycles (only one choice above).
3. Add unifiable connections necessary to form a spanning set (the two additional connections above).

4. Select appropriate "base" connections (the $FAC(0,1)$–connection above) providing the initialization for the variables involved.

5. Extract the cycle equations (as done above).

6. Possibly normalize these equations (not needed above; for an example see next section).

7. Extract the termination equations from the remaining connections (as done above).

8. Transform the resulting set of equations into a bottom–up destructive assignment algorithm (the result was shown above).

Selections in this algorithm have to be interpreted as non–deterministic features that have yet to be transformed into some sort of a while–loop (or, in other words, some sort of backtracking). The present example is so simple that no alternatives are possible.

There is an alternate way of looking at the equations resulting from the unification of the connected terms which will be helpful as a guide to the more refined version of the algorithm. Namely, the cycle's effect may be seen as that of a function f that takes a pair (x,y) as input and returns the resulting pair $f(x,y)$ as output. Let us call f the *cycle function* associated with the cycle in question (a way to determine and represent such a function will be shown shortly). This observation holds for any cycle although one would have to consider arbitrary tuples rather than just pairs in the general case.

If the cycle is traversed k times then the output will be $f^k(x,y)$. If the initial values for (x,y) have been obtained in step 4, viz. $(0,1)$ in the present case, then step 7 provides a single equation that is $(n,z) = f^k(0,1)$ in our case. Actually this is a vector equation and therefore consists of two equations, one for each component. Since this makes two equations with two unknowns, k and z, they can be solved, that is, z may be provided in a functional form dependent on the input only. So z may as well be computed in a functional way which is what we are aiming for.

The extraction of f is particularly simple in our case where the pair (x,y) is explicitly contained in the subgoal (i.e. without any further function symbols) and where the cycle contains only a single connection. In this simplest case f can just be read off from the other literal in the connection. That is, here the function consists of a pair where the left component is the $+1$–function and the right one the $\cdot$ –function with appropriate argument–functions. Using the primitive functions and operations from recursion theory and following the

notation in [13], then for the factorial program the full equation is the following.

$$(n, z) = [(\varsigma \circ \pi_2) \times (\cdot \circ < \varsigma\,,\ \pi_2 \circ\ {}_1\xi_2 >)]^{\#}((0,1), k)$$

Since $n = \varsigma^{\#}(n)$, it is straightforward to determine k and z in this case. The reader not familiar with this functional language should not worry too much. The only lesson to be learned from this exercise is that the compilation of recursive cycles indeed boils down to solving equations. If we take the present alternative way of determining the cycle function then step 8 in the algorithm becomes even superfluous since the primitive functions and operations used on the right side of the equation above provide a perfect and highly efficient programming language.

Note that the resulting functional program in the present example is iterative in the sense of a DO–loop rather than a WHILE–loop. So the performance of the resultant program is even better than the algorithmic program shown further above (by using a register rather than a termination test to be executed in each cycle). This may be achieved in general following this approach as long as we are computing primitive recursive functions which is done anyway most of the time.

Coming back to the algorithm itself, let us emphasize once more that we are not claiming to handle *all* recursive problems this way, of course. The point is that we aim at an identification of a class that consists of a rather simple, but popular kind of problems to be treated this way. Simple problems in this sense are those that have *linear* cycles only. In fact the present version obviously has only a single *cycle* in mind. Another feature that makes problems simple is given when the output tuple (the pair (n, z) in the present example) may be represented in closed form as in the equation above which of course is not always possible (the Ackermann function is the classic example). We will refer to this as to the *solvability* feature.

Note that the approach taken here with compiling logic programs is different from what is usually considered in the compilation of logic programs. There, no transformation into an iterative form takes place as here. Of course, the program might be presented in an iterative way by the user himself (see programs 8.3 and 8.4 in [23] for computing the factorial in an iterative way in PROLOG). Although possible, we would *not* like to have the programmer bother about this issue.

The transformation we suggest is also not provided by the technique of tail recursion optimization used in PROLOG. This optimization re–uses the memory area allocated for the parent goal for the new goal if this is the last call in the body of a rule. Although this technique can be used in our program above it does not provide a cure to the problem under discussion. The memory

requirements still grow linearly with n, since the result variable z has to store the intermediate symbolic results $n \cdot y$, $n \cdot (n-1) \cdot y$, etc., whereby y cannot be evaluated. In addition there are the extra efforts needed for building each of the frames associated with the procedure calls. The fact that here this growth could actually be prevented by computing the result of the subexpressions like $n \cdot (n-1)$ is accidental with this particular example, and would not apply in all cases where our approach would still work. In summary, we claim that the compilation techniques used in PROLOG could be enhanced by the bottom–up compilation we are proposing here.

4.3 The incorporation of factoring

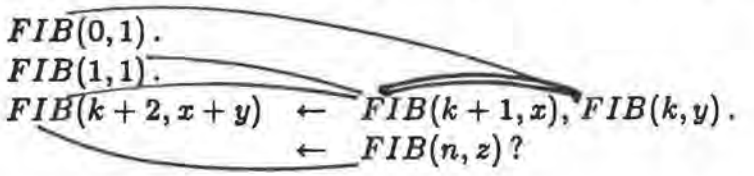
The factorial problem is too simple to illustrate all the features that have to be taken into account in our approach. Therefore we analyze a more complicated example in the present section, viz. the Fibonacci problem, and test the previous algorithmic steps with it. Thereby we prefer the more illustrative first alternative of representing the equations resulting from the unifications.

$$
\begin{array}{lcl}
FIB(0,1)\,. & & \\
FIB(1,1)\,. & & \\
FIB(k+2,x+y) & \leftarrow & FIB(k+1,x),\ FIB(k,y)\,. \\
 & \leftarrow & FIB(n,z)\,?
\end{array}
$$

There are two connections in the rule that may both, separately or together, be used to form a recursive cycle. If we consider using them both for a single cycle then this one would no more be linear (hence much more complicated) since the head literal would occur in both connections of the cycle. So we try to succeed by using only one of them and select the one containing the rule's first subgoal (the other alternative being briefly mentioned later). This completes the first two steps of the compilation algorithm.

Step 3 requires determining a spanning set of connections. Ignoring the trivial alternatives that do not involve the cycle connection, we proceed in a mechanical way as follows. There is only a single connection possible that involves the rule head and does not interfere with the cycle, namely the one with the program's goal; so this must be taken. Similarly, there is no choice with the first subgoal which only unifies with $FIB(1,1)$. This literal is also contained in the connection with the other subgoal to be taken into consideration. $FIB(0,1)$ is sort of an "isolated literal" since for the spanning set of connections to be assembled there is no other connection that contains this literal. It must be selected as the connection required for $FIB(k,y)$ since otherwise unifiability fails for one of the two connections that contain $FIB(1,1)$.

This selected set of connections is now spanning except when the cycle connection is involved since then the second subgoal literal is pure in instances other than the first two ones (not unifiable anymore with $FIB(0,1)$ and $FIB(1,1)$). This is where this problem differs significantly from the previous ones. At this point we recall the factoring technique in Automated Theorem Proving [3], that is, we consider factoring the first and second subgoals in the rule (illustrated as a factoring connection) as the only possibility left to settle the spanning property requirement. So altogether we arrive at the following situation.



Step 4 in the algorithm now yields the equations $k_1 = 0$, $x_1 = y_1 = 1$. The equations $k_2 = 1$, $y_2 = 1$ in a refined version of the algorithm might result not before step 5 has been performed since only then this assignment becomes obvious to a mechanical procedure. Step 5 yields $k_{i+1}+1 = k_i+2$ and $x_{i+1} = x_i + y_i$, where the first one in step 6 will be reduced to $k_{i+1} = k_i + 1$. This equation now determines the instances that are linked together by the factoring connection. Hence in addition we obtain $y_{i+1} = x_i$. From now on the situation is the same as in the factorial problem so that we may proceed exactly as demonstrated there with performing the remaining steps 7 and 8. In other words, our compilation algorithm succeeds also in solving this more complicated problem if factoring is included in the proof technique as it certainly should.

Above we mentioned the other alternative of selecting a linear cycle. Had we selected this one then the resulting equation $k_{i+1} = k_i + 2$ would turn out to be inconsistent with the constraints resulting from the factoring connection, which discards this alternative.

4.4 A datalogic example

As we mentioned before, a lot of work has already been done to compile recursion for the access of a database. By treating the transitivity problem frequently studied in that area we first want to demonstrate the relevance of our algorithmic approach to this sort of application. A more detailed evaluation of the relationship with that area and its terminology will then follow.

$$
\begin{array}{l}
Aab. \\
Abc. \\
Acd. \\
Ade. \\
Axz \leftarrow Axy,\ Ayz. \\
\qquad \leftarrow Avw\,?
\end{array}
$$

Before we apply our algorithm from Section 4.2 to this example we clarify the circumstances under which it is meant to be applied. One alternative would be to run this algorithm at compile time with the query already available. Another one would consider the compilation prior to the queries being available. It actually makes quite a difference which of these two alternatives is assumed since in the first case we would have the information about the specific input, its argument position and its value, while in the second one neither are known. For real applications we have to provide solutions for both of these situations. Prior to any query available the database has to be organized for a fast access via the recursive rule. But once the query is available its particular features should be used once more to speed up the access even further. In the following we thus first consider the case where it is not known which of v and w actually carries the input and what this input is.

As a second remark note that it is not this particular problem for which we strive to provide a particularly smart database solution. Rather our algorithm (in its refined version) is meant to handle any such problem similarly well, obviously a much more ambitious goal. In the present case the result should be that for a given input, say on v, the compiled algorithm would just start at the appropriate point in the list (c, d, e) to just read off all the possible outputs from the remainder of that list. With these preliminaries in mind let us now see how our algorithm would proceed with this particular program whereby a number of issues will be mentioned that will have to be incorporated in the envisaged refined version of the algorithm.

The program has a structure very similar to that of the Fibonacci program. So if we again restrict our considerations to linear cycles then the first step produces the two obvious alternatives from which we select (in step 2) the first one. In order to put together a spanning set of connections in step 3, the technique of factoring from the previous section would be naturally attempted in this case again. It would fail, however, in the present case, since the resulting unifications would transform our cycle into a tautological one (so a test for tautological cycles as a consequence of unifications being, for instance, among the necessary refinements of the algorithm mentioned above). The only way to assemble a spanning set is by solving the second subgoal in the rule in each

instance with a fact from the database. All other connections are then obvious. So we obtain the following equations (step 5).

$$x_{i+1} = x_i = v\,,\ y_{i+1} = z_i\,,\ z_{i+1} \in \lambda u A z_i u\,,\ w = z_k$$

There are two cases to be considered. Assume the first where v carries the input which in step 7 yields

$$y_1 \in \lambda u A v u\,,\ z_1 \in \lambda u A y_1 u\,,\ w = z_i\,,\ i \in \{1,2,\ldots\}$$

So indeed the algorithm provides a precomputation for this case so that for a concrete v the resulting w–values may just be read off the z–list assuming that a further refinement of the algorithm has provided the maximal list which is (c,d,e) as noted before.

The other case where w carries the input goes similarly. Both cases may be prepared by the compiler without having the actual input. Once the input is available the appropriate choice will be made. The second alternative in the choice of the two cycles by symmetry leads to the same solutions, an insight that at this point we would not expect from the compiler in mind (rather it would – redundantly – handle it as an alternative).

In summary, it turns out that our proposed algorithm provides the adequate framework for this kind of problem as well.

4.5 A logical view of datalogic

A *database* is a collection of explicitly given data organized in some way (see [22] for a more detailed definition). There has been an agreement on this notion for decades; only recently various authors are producing notational confusion that we will try to clarify in the following.

There is no need for a definition of *logic*, since it is well–known, not only for decades but even centuries. *Horn clause logic* is a more recent concept (denoting a certain part of logic) but still predates the emergence of the concept of databases. *PROLOG* is a programming language based on Horn clause logic. From a logical point of view databases are concerned with that part within (Horn clause) logic, that concerns the facts (i.e. the clauses consisting of a ground literal) only.

Given this common ground it is natural that there has always been an interaction between logic and the database field; let us just mention three logical papers [16,10,22], from three rather different periods, with an emphasis on databases. This interaction has recently intensified considerably so that it is justifiable to speak of a whole area in its own right. In [20] the term *datalog*

is used that we slightly modify to the more elegant *datalogic*; either of these two terms is adequate since there is no doubt that we are dealing with (Horn clause) *logic* once rules are added, but at the same time the emphasis within logic is on the *data*.

Other names such as deductive databases (or even simply databases [1]) are ill–conceived. Since there is only the most trivial form of deduction possible in a database, which is the matching of a query (typically a literal) with a fact (another literal) in the database, what could be "deductive" about it? This remains true even though some deductions may be compiled into database or relational algebra operations as we have shown further above. It is not the database operations themselves that gain more power by this compilation, rather it is the compilation that in certain cases encodes the deductive steps with a combination of such simple operations.

The features behind the important notions of datalogic are quite familiar in the literature of Automated Theorem Proving for many years. For instance, [6,3] survey the origins of the notion of a cycle in the early seventies and even before. As we have shown above, the cyclic feature is the essence of a recursive rule. It should therefore be simple to restate the definitions of some datalogic notions (cf. [1,20]) in terms of cycles, a fact that will be demonstrated now. Of course, all the following definitions are restricted within datalogic to the Horn clause case while the cycle notions actually apply to the general case.

Definitions A predicate P is said to be *recursive* if there is a potentially recursive cycle containing P (i.e. P appears in a literal in some of its connections). Two predicates P and Q are called *mutually recursive* if there is a potentially recursive cycle containing both P and Q. A rule in a set of rules is called *recursive* if there is a potentially recursive cycle containing the rule's head. A recursive rule is *linear* if all potentially recursive cycles containing its head are linear. A set of rules is *linear* if all its recursive rules are linear. Two rules are *mutually recursive* if there is a potentially recursive cycle containing both their heads.

All these so–defined notions coincide with those in datalogic at least in spirit, but not always in the fine details. For instance, the two rules

$$\begin{array}{l} Px \leftarrow Qx. \\ Qy \leftarrow Py. \end{array}$$

contain a tautological cycle which therefore is not potentially recursive; hence the two rules are not mutually recursive in our terminology while they usually fall into that category in datalogic. Since the cycle here truly is a tautological, not a recursive one, we prefer our refined notion. Also note that a non–linear

rule may still be treated in a linear way with linear cycles, as we have demonstrated with the Fibonacci and transitivity examples above, which once again shows an advantage of our refined terminology.

On the other hand we do not at all claim that this short subsection covers all the relevant techniques for treating recursion in the context of a database with maximal efficiency. For instance, concepts such as *adornments*, *magic sets*, *counting*, and others [1,20] would have to be discussed in a more comprehensive survey. Nevertheless we are convinced that these might preferably be incorporated on the basis outlined in the previous sections whenever they add truly new concepts to those introduced above.

4.6 Limitations

The technique for treating recursion efficiently, that has been described in the last few subsections, will, of course, not succeed in all cases. On purpose we have restricted ourselves to the treatment of a limited class of problems. This class is determined by the following two characteristics.

A Only linear and single cycles are considered, that feature

B the solvability property (see Section 4.2).

This limited class clearly contains all the tail recursion problems that have been mentioned in Section 4.2. On the other hand, there are many recursive problems that cannot be treated with our technique. An example is the following quite complex non–Horn problem [21].

$$
\begin{array}{lcl}
Pxz & \leftarrow & Pxy,\, Pyz\,. \\
Qxz & \leftarrow & Qxy,\, Qyz\,. \\
Qyx & \leftarrow & Qxy\,. \\
Pxy \vee Qxy\,. & & \\
 & \leftarrow & Pab \vee Qcd\,?
\end{array}
$$

This formula allows nearly no reductions of the sort we mentioned so far. It has many different cycles and gives no indication which of them should be preferred in order to find the proof. In fact the proof is made up by a recursive cycle that involves all possible connections; so in any case the cycle is highly non–linear. Moreover, this cycle is traversed by the proof in a way that does not suggest any obvious regularity and further requires three instances of each rule except the first one where two suffice. So we would say, its apparent non–obviousness (just try to proof it by hand) is manifested by these characteristics: non–reducibility, non–linearity (or –regularity in a more general sense), and

need for a relatively high number of instances. In the early work of the present author the last property was taken as the only characteristic parameter (see the *degree* in [2]). With the deeper insight gained over the years we now propose a more refined "degree" of non–obviousness that takes into account the other two parameters as well. To formalize such a more complicated degree in a logically convincing way might not be that easy a task, though.

This example illustrates that there are formulas that need non–linear cycles to be proved. So the restriction to linear cycles properly limits applicability. On the other hand it is a very natural restriction worth exploring, since "most" practical problems may be treated this way (whatever this means).

From a theoretical point of view one would like to have a syntactic characterization of the class of problems that may be treated with our technique, such that for a given formula it can be determined whether it is a member or not (like the characterization of Horn formulas). So far we can only offer the characterization that is provided and may be applied by carrying out the proposed algorithm. Since the algorithm is reasonably efficient, we think that this characterization is actually good enough for practical purposes, although it certainly would be useful if someone in Theoretical Computer Science would find a more explicit one.

In [8] we have presented an alternate way of functionalizing a logic program that takes into account an appropriate induction scheme. It seems that the proposal made here is more direct, at least for relatively simple programs. The other proposal might then be useful in cases where a linear cycle does not provide the solution, i.e. in more complicated cases. But a more elaborate comparison has yet to be carried out.

5 Quantifiers and modal operators

Logic will not forgive inaccuracy while people are fond of their little mistakes. This is one reason why logic has not really attained popularity to an extent it certainly should. Being the single available formalism capturing essential features of human thought it should be taught at elementary schools already in some way or another. But even Mathematics professors question any relevance of logic to their work.

Logic has now gained a little popularity through PROLOG particularly for applications in the knowledge–based systems area. As we have seen PROLOG is limited to the simple logical structure of rules. It is interesting to watch the field, how it currently stretches its limbs in order to try to fit into the jacket that obviously is too tight. Negation is one cause for discomfort, quantifiers are

another.

Indeed quantifiers are never mentioned in PROLOG. Nevertheless they are present by the nature of human logic. We just do not mention them explicitly (for the psychological reason mentioned above), assuming a default structure in this respect instead. Unfortunately, there are even very practical examples where the original default jacket simply does not fit. [20] mentions some of these and proposes an improved default instead. While this is fine, the principal problem does not disappear which is that quantifiers are essential and cannot be defined away. They abound in natural language in particular. So we have to deal with them in a conscious way, even in the default case. Let us have a brief look at this default case.

It is achieved by skolemization (see [3]). Instead of saying "Everyone has a father" which all–quantifies by way of the "everyone" and existentially quantifies by way of the "a", a "father"–function is introduced which substitutes the existential quantifier. This way only all–quantifiers are left which then are ignored by default. There are a number of reservations with this solution.

First of all, the meaning actually has changed slightly by this transition. While the natural statement did not rule out the possibility of having more than one father, the functional version actually does by the nature of functions. Second, in a nested quantificational structure the skolemization gives up quite a bit of information about this structure retaining only the relative position of the existential quantifiers with respect to the all–quantifiers. Thirdly, the length of the formula may increase by the transition, quadratically in the worst case which in turn causes an increased overhead in the performance. Finally, in view of advanced unification techniques skolemization actually turns out to be a redundant step. More on these points can be found in [3].

Raising these issues means begging for alternatives. As just indicated, the main function of skolemization is to record the relative occurrences of the different types of quantifiers within a formula. The proof technique most sensitive for the position of any logical operation within a formula is that based on the Gentzen–type natural deduction systems. And indeed an analysis of these systems has provided such an alternative. Rather than introducing Skolem functions, this alternative achieves the right way of unification by taking into account the structure of the relation that is provided by the operational symbols within the formula the way they occur in relation to each other. While there have been other approaches to the automation of deduction on the basis of calculi of natural deduction, this one is unique insofar as it does not dispense with the concern for efficiency comparable to that of resolution. An outline of this whole approach is given in [5]. For a comprehensive treatment the reader has to consult [3].

The technique used in this approach is not dependent on the logical operators involved. So in a sense it does provide an efficient treatment of deduction in (to some extent) *any* logistic calculus. [3] further presents the treatment of higher–order logic this way. Very recently now Wallen [26] has picked up these ideas and applied them to various modal logics, specifically to the modal systems K, K4, D, D4, T, S4, and S5. He succeeded in providing efficient proof mechanisms for all these systems. Actually, since the technique is as general as it is, one may expect its successful application to other systems as well along exactly the same lines. Since we did not want to spoil the beauty of Wallen's work by giving a sketchy overview of it, the remainder of this section will just state a few observations in this context. The interested reader might find it extremely rewarding to read the original source itself, perhaps start with [27] as an appetizer.

The work is done on the basis of a sequent calculus [5]. All the modifications needed to extend the classical calculus to cope with these modal systems still preserve the property of being cut–free, which is of vital importance for the automation of deduction. Also they retain the subformula property and, of course, the basic structure of the sequents.

One central part in establishing provability is an analysis of the propositional structure of the given modal formula. In fact, it turns out that this part is in essence identical with the analogue part in first–order logic. That is, a spanning set of connections has to be identified. What is different are the conditions under which a pair of literals can be deemed complementary.

The key observation is that an appropriate notion of complementarity can be defined by noting the context of atoms relative to the modal operators in the endsequence, and considering mappings of representations of these contexts, called prefixes, which render the prefixes of the components of connections identical. The first part of this is similar to incorporating the relation among occurrences of logical operators into the unification process, while the second part, which makes it actually work, is an original contribution of extreme elegance. These mappings can in fact be seen as substitutions in the usual sense. Their exact definition depends on which modal system we are dealing with.

Testing a formula for validity in any of these modal logics thus is in a sense the same as in classical logic, so that any kind of search strategy developed so far is applicable without alteration. For instance, all the material presented in the previous sections have full relevance to modal logic this way. What changes is the unificational part, which, for instance, does not always produce a single most general unifier, although the set of such unifiers is always finite.

It should be noted that this seems to be the first proof mechanism for such modal systems that is both fully general and (comparatively) efficient. Its

compatibility with first–order proof techniques makes it even more attractive. One has to get serious with quantifiers and modal operators, though, in order to appreciate this beauty.

6 Building–in theories

All the material in the previous sections focused on the purely logical part of deduction. For practical purposes it is essential though that in a certain context advantage is taken of deductive structures that occur in a stereotype way again and again (resulting from the theory defined by the context). Equality has to be treated this way to mention just one example. The previous volume contains much material on this topic [18,24]. This is the only reason why no additional attention is given to this area here, so the reader should not draw false conclusions about the importance of this area.

In addition there has been some progress in the meantime with the treatment of mathematical induction in the context of a theorem prover based on the connection method reported in [14]. There a special sort of connection is introduced that links literals if they reoccur in a proof with an inductive nature. A system based on this approach is actually operative.

The restriction to literals in this inductive setting seems not be that harmful since more complicated formula parts may be abbreviated with literals by inclusion of appropriate definitions as also Wallen (private communication) suggests.

7 Conclusions

This chapter is meant to provide an update on the material contained in a previous volume [11] on the topic of Automated Deduction. As has to be expected for such a wide and active area, only a few major issues could be discussed at some length and to some degree of detail here. We have chosen these to be the following ones.

A bottom–up reduction technique has been suggested that enhances the performance of top–down theorem provers such as PROLOG systems. It is well–known that bottom–up executions may quickly result in an exponential combinatorial explosion. This is avoided here by carefully restricting the bottom–up execution to those deductive parts of the program that are guaranteed to be settled in linear time. These parts are identified in a syntactic way with the notion of isolated connections along with other preparatory features like DB–reductions. It is felt that this kind of technique has a great importance, in particular for practical applications.

The program resulting from these operations typically is recursive in nature. Such recursive programs have been analyzed in some detail, in particular by way of the basic concept of a (deductive) cycle. On the basis of this analysis the framework of an algorithm is given that achieves the compilation of a recursive logic program into code of a purely iterative nature (basically a DO–loop), at least for a class of practically important cases. This algorithm is illustrated with selected examples demonstrating key issues such as the need for factorizing and the incorporation of databases.

The remaining two sections are more of the nature of providing pointers to material of particular interest in the literature, such as an extremely elegant approach to the automation of deduction in modal logic.

References

[1] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. SIGMOD Conf. on Management of Data*, pages 16–52, ACM, 1986.

[2] W. Bibel. An approach to a systematic theorem proving procedure in first-order logic. *Computing*, 12:43–55, 1974.

[3] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, second, 289 pages edition, 1987.

[4] W. Bibel. *Constraint Satisfaction from a Deductive Viewpoint.* Technical Report, Forschungsgruppe Künstliche Intelligenz, Technische Universität München, 1987 (submitted).

[5] W. Bibel. Methods of automated reasoning. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence — An Advanced Course*, pages 173 - 222, Springer, LNCS *232*, Berlin, 1986.

[6] W. Bibel. On matrices with connections. *Journal of ACM*, 28:633–645, 1981.

[7] W. Bibel. Prädikatives Programmieren. In *GI — 2. Fachtagung über Automatentheorie und Formale Sprachen*, pages 274–283, Springer, Berlin, 1975.

[8] W. Bibel. Predicative programming revisited. In W. Bibel and K. Jantke, editors, *MMSSSS'85 — Mathematical Methods for the Specification and Synthesis of Software Systems*, pages 24–40, Springer, Berlin, 1986.

[9] W. Bibel. *Programmieren in der Sprache der Prädikatenlogik.* (Rejected) thesis for "Habilitation" presented to the Faculty of Mathematics, Technische Universität München, January 1975.

[10] W. Bibel. *A Uniform Approach to Programming.* Bericht 7633, Technische Universität München, Mathematische Fakultät, 1976.

[11] W. Bibel and Ph. Jorrand, editors. *Fundamentals of Artificial Intelligence. Study Edition*, Springer, Berlin, 1987.

[12] W. Bibel, R. Letz, and J. Schumann. Bottom-up enhancements of deductive systems. In I. Plander, editor, *Proceedings of 4th International Conference on Artificial Intelligence and Information-Control Systems of Robots*, North-Holland, Smolenice, CSSR, October 1987.

[13] W. S. Brainerd and L. H. Landweber. *Theory of Computation.* Wiley, New York, 1974.

[14] M. Breu. *Einbeziehung einfacher Induktionsbeweise in den Konnektionenkalkül.* Diplom-thesis, Technische Universität München, 1986.

[15] E. Eder. Properties of substitutions and unifications. *Journal for Symbolic Computation*, 1:31–46, 1985.

[16] C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, Edinburgh University Press, 1969.

[17] L. J. Henschen and S. A. Naqvi. On compiling queries in recursive first-order databases. *Journal of ACM*, 31:47–85, 1984.

[18] G. Huet. Deduction and computation. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence — An Advanced Course*, pages 39–74, Springer, LNCS *232*, Berlin, 1987.

[19] J. Minker, D. H. Fishman, and J. R. McSkimin. The Q^* algorithm — a search strategy for a deductive question-answering system. *Artificial Intelligence*, 4:225–243, 1973.

[20] K. Morris, J. D. Ullman, and A. van Gelder. Design overview of the NAIL! system. In E. Shapiro, editor, *Proc. 3rd Intern. Conf. on Logic Programming*, pages 554–568, Springer (LNCS *225*), Berlin, 1986.

[21] F. J. Pelletier and P. Rudnicki. Non–obviousness. *AAR Newsletter*, 6:4–5, 1987.

[22] R. Reiter. Towards a logical reconstruction of relational database theory. In M. L. Brodie et al., editor, *On Conceptual Modeling*, pages 191–238, Springer, Berlin, 1983.

[23] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge MA, 1986.

[24] M. E. Stickel. An introduction to automated deduction. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence*, pages 75–132, Springer, Berlin, 1987.

[25] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville MD, 1982.

[26] L. Wallen. *Automated Deduction in Modal Logics.* PhD thesis, University of Edinburgh, 1987. PhD Thesis.

[27] L. Wallen. Matrix proof methods for modal logics. In *IJCAI'87*, pages 917–923, Morgan Kaufmann, Los Altos CA, 1987.