ELSEVIER

Artificial Intelligence 90 (1997) 135–176

Artificial Intelligence

Computing answers with model elimination ★

Peter Baumgartner*, Ulrich Furbach[1], Frieder Stolzenburg[2]

Department of Computer Science, University of Koblenz, Rheinau 1, D-56075 Koblenz, Germany

Received November 1995; revised April 1996

Abstract

We demonstrate that theorem provers using model elimination (ME) can be used as answer-complete interpreters for *disjunctive logic programming*. More specifically, we introduce a family of restart variants of model elimination, and we introduce a mechanism for computing answers. Building on this, we develop a new calculus called *ancestry restart ME*. This variant admits a more restrictive regularity restriction than restart ME, and, as a side-effect, it is in particular attractive for computing definite answers. The presented calculi can also be used successfully in the context of *automated theorem proving*. We demonstrate experimentally that it is more difficult to compute nontrivial answers to goals than to prove the *existence* of answers. © 1997 Elsevier Science B.V.

Keywords: Automated reasoning; Theorem proving; Model elimination; Logic programming; Computing answers

In first-order automatic theorem proving one is interested in the question whether a given formula follows logically from a set of axioms. This is a rather artificial task; whenever one is interested in solving real problems it is often necessary to compute answers for given questions. It appears to us, that in automated theorem proving this aspect has been pushed to the background. For instance, the statements of the puzzle problems in the TPTP library [33] also include their solutions in most cases, and the prover has only to *verify* them; however, it is much more interesting and more difficult to *find* a solution than to prove correctness of a given one.

★ This research was sponsored by the "Deutsche Forschungsgemeinschaft (DFG)" within the "Schwerpunktprogramm *Deduktion*". A predecessor version of this paper appeared in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montréal, Que (1995) 335–340.

* Corresponding author. E-mail: peter@informatik.uni-koblenz.de.

[1] E-mail: uli@informatik.uni-koblenz.de.

[2] E-mail: stolzen@informatik.uni-koblenz.de.

PII S0004-3702(96)00042-2

In the early days, when automated theorem provers were understood as tools for real world problem solving, this problem was apparent: of course the textbook monkey was not interested whether *there is* a solution of the monkey and banana problem; it was merely interested in finding a way to reach the banana hanging on the ceiling. In [8] there is a whole chapter on question answering, where, for example, the work of Green on question answering [13] is reviewed thoroughly. We feel that this aspect is not given sufficient attention in modern theorem proving literature.

In the logic programming area, the computation of answers has always been an important aspect. From there we learned that it is more difficult to prove a calculus to be answer-complete than only refutationally complete. Recently, there has been considerable effort to use full first-order logic instead of only Horn clauses as the basis of logic programming [21]. This (positive) disjunctive logic programming approach is investigated from various directions. From the view of theorem proving one is concerned with modifying theorem provers such that they can be used as interpreters for logic programming purposes. The nonmonotonic reasoning community is working on finding appropriate semantics for disjunctive logic programs with negation, and from a database viewpoint one is concerned with finding bottom-up approaches to computations.

The aim of this paper is twofold: Firstly, we prove that theorem provers using model elimination (ME) can be used as answer-complete interpreters for disjunctive logic programming. Secondly, we demonstrate that in the context of automated theorem proving it is more difficult to compute nontrivial answers to goals than to prove the existence of answers. We furthermore investigate mechanisms for finding most specific answers which give us the most concise information. Technically speaking this means, we are looking for the most definite or shortest disjunctive answers with the most general variable substitutions.

Concerning the first aspect, it is important to note that there is a lot of work towards model theoretic semantics of *positive disjunctive logic programs*, and of course there are numerous proposals for nonmonotonic extensions. However, with respect to proof theory, the situation is not so clear. At first glance one might be convinced that any first-order theorem prover can be used for the interpretation of disjunctive logic programs, since a program clause $A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n$ is a representation of the clause $A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_n$. The notions program and clause will be defined formally in Section 1.2. Indeed, in [18] SLI resolution is used as a calculus for disjunctive logic programming. From logic programming with Horn clauses, however, we learn that for a procedural interpretation of program clauses it is crucial that clauses can only be accessed by the literals A_i, i.e., by the head literals. Technically, this means that only those contrapositives are allowed to be used which contain a positive literal in the head. The approach from [18] completely ignores this aspect by using SLI resolution which requires all contrapositives.

There are proposals for first-order proof calculi using program clauses in only this procedural reading, e.g. Plaisted's problem reduction formats [27], or the near-Horn Prolog family introduced by Loveland and his co-workers [22]. For a thorough discussion we refer to [4]. These approaches introduce new calculi or proof procedures, for which efficient implementations still have to be developed. Our aim is to modify ME such that it can be used for logic programming in the above sense. We introduce various

refinements of restart model elimination and finally arrive at a calculus which supports a procedural reading of first-order program clauses.

This gives us the possibility to use existing theorem provers for disjunctive logic programming. As a first step towards this goal, we introduced in [4] the restart variant of ME and proved its refutational completeness. In this paper, we restate restart model elimination together with its variants. We furthermore give a ground completeness proof of its weakest variant, which gives us the basis of properly introducing an answer computing mechanism into restart model elimination (RME). Furthermore we define a variant called *ancestry RME* which allows extended regularity checking (i.e., loop checking) w.r.t. the ordinary RME. Additionally this variant prefers proofs which permit definite answers.

For the second aspect, namely *computing answers*, we adapted our implementation of the RME variants—the PROTEIN system [5]—for answer computing as described below. We demonstrate with some of Smullyan's puzzles [30] that it is more difficult to compute answers than to prove unsatisfiability. Finally we give a comparative study of high performance theorem provers, including OTTER [25], SATCHMO [24], SETHEO [16], and PROTEIN [5].

1. Model elimination and the restart variants

Model elimination, as originally introduced by Loveland in [19], plays a central role in our approach. We will introduce various variants, and especially we will use as a formal starting point the restart model elimination calculus as introduced in [4]. Furthermore we will discuss how the calculus can be used for logic programming purposes by introducing refinements.

In the following subsection we informally describe a tableau-based variant of model elimination for the propositional case, and in the rest of this section we introduce our calculi in a formal way for the first-order case.

1.1. Tableau model elimination

In this subsection we use the clause notation, mirroring the fact that we review a calculus which is, as it stands, not suited for programming purposes. We use an ME calculus that differs from the original one presented in [19]. It is described in [16] as the base for the prover SETHEO. In [3] this calculus is discussed in detail by presenting it in a consolution style [11] and compared to various other calculi. ME, in this sense, manipulates trees by extension and reduction steps.

Example 1. In order to recall the calculus consider the clause set

$$\{P \vee Q, \neg P \vee Q, \neg Q \vee P, \neg P \vee \neg Q\}.$$

A model elimination refutation is depicted in Fig. 1 (left side). It is obtained by successive fanning with clauses from the input set (*extension steps*). Additionally, it

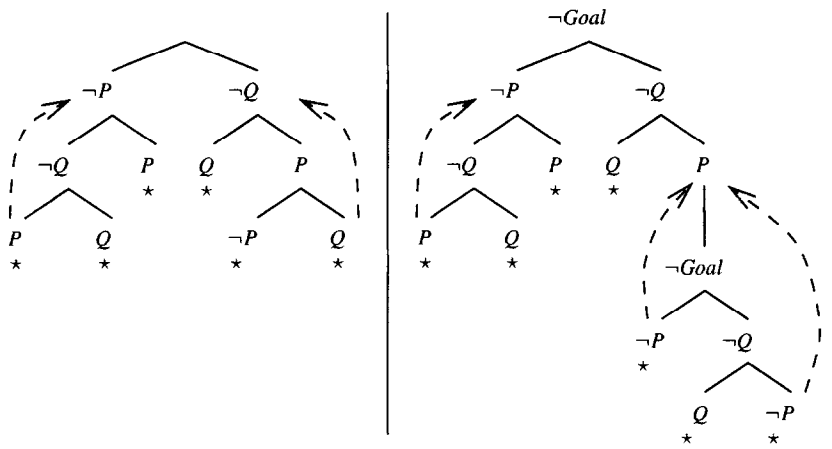

Fig. 1. Model elimination (left side) versus restart model elimination (right side). In order not to overload the notation, positive *Goal* nodes are not displayed.

is required that every inner node is complementary to one of its sons. Such sons are annotated with a "$\star$" in Fig. 1. A dashed arrow indicates a *reduction step*, i.e., the closing of a branch due to a branch literal complementary to the leaf literal. Extension and reduction steps are allowed at any leaf of the tree and for extension steps any literal from an input clause can be used to form a complementary pair of literals. For example, in the right subtree of Fig. 1 (left side) the clause $\{\neg P, Q\}$ was used to extend the positive leaf P, i.e., we used the program clause $Q \leftarrow P$ via the body literal P and hence dispensed with a procedural reading of the clause.

Now we show how restart model elimination treats Example 1. As a preliminary step, the input clause set has to be transformed into what we call *Goal normal form*: every purely negative clause is conjoined with the new literal *Goal*. Thus, in the example, we replace $\{\neg P, \neg Q\}$ by $\{Goal, \neg P, \neg Q\}$. Further, as query the unit clause $\neg Goal$ is used. Fig. 1 (right side) displays a restart model elimination refutation. Besides using the goal normal form, the only difference is that extension steps at positive literals are not allowed; instead either a reduction step is carried out, or else the root literal—which is always $\neg Goal$—is copied, and then an extension follows.

These simple modifications obviously allow only extension steps with a positive, i.e., a head literal of a clause, and hence support a procedural reading of program clauses. In the following subsection we give a formal presentation of the calculus along the lines of [3].

1.2. Restart model elimination (RME)

We will state some basic definitions. A *clause* is a multiset of literals, usually written as the disjunction $L_1 \vee \cdots \vee L_n$. Clauses can be alternatively represented with an arrow. $A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n$ is a representation of the clause $A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_n$, where the A and B are atoms. Clauses with $m \geqslant 1$ are called *program clauses* with

head literals A_i and *body literals* B_i, if present. Clauses of the form $\leftarrow B_1 \wedge \cdots \wedge B_n$ are called negative clauses in the sequel.

In Section 3 below we will restrict clause sets to *programs*, which consist of program clauses only; but this distinction is not needed now. However, from now on we assume our clause sets to be in *Goal normal form*, i.e., there exists only one negative clause which furthermore does not contain variables. Without loss of generality this can be achieved by introducing a new clause $\leftarrow$ *Goal* where *Goal* is a new predicate symbol, and by replacing every purely negative clause $\neg B_1 \vee \cdots \vee \neg B_n$ by $Goal \leftarrow B_1, \ldots, B_n$.

We are now turning towards the calculus. Throughout this paper, we consider a variant of ME which uses so-called ME tableaux as basic proof objects, see [15], rather than ME chains [20].

Definition 2 (*Literal tree, branch, etc.*). A *literal tree* T is a pair (t, λ) consisting of a finite, ordered tree t and a labelling function λ that assigns a literal to every node of t. Nodes are classified as *positive* or *negative*, as given by the polarity of the assigned literals. If δ is a substitution, then $T\delta$ denotes the literal tree which is obtained from literal tree T by application of δ to the labels of all nodes of T. That is, if $T = \langle t, \lambda \rangle$ then $T\delta = \langle t, \lambda' \rangle$, where $\lambda'(N) = (\lambda(N))\delta$ for every node N in T.

A branch of length n for a given literal T is a sequence $[N_0 \cdot N_1 \cdot \ldots \cdot N_n]$ ($n \geqslant 0$, written as indicated) of nodes in T such that N_0 is the root of T, N_i is the immediate predecessor of N_{i+1} for $0 \leqslant i < n$, and N_n is a leaf of T; the functions *First* and *Leaf* return the first, respectively last node of a branch, i.e., $First([N_0 \cdot N_1 \cdot \ldots \cdot N_n]) = N_0$ and $Leaf([N_0 \cdot N_1 \cdot \ldots \cdot N_n]) = N_n$.

Throughout this paper, the letter N is used for nodes, and the symbols p, q are used for branches; like branches, branch-valued variables are also written with brackets, as in $[p]$.

For our purposes it is useful also to use the branch representation of a literal tree $\langle t, \lambda \rangle$. That is, we identify $\langle t, \lambda \rangle$ with $\langle \boldsymbol{P}_t, \lambda \rangle$, where $\boldsymbol{P}_t = \{[p] \mid [p] \text{ is a branch of } t\}$. If context allows we simply write $\boldsymbol{P}$ instead of $\langle \boldsymbol{P}_t, \lambda \rangle$. Branch sets are typically denoted by the letters $\boldsymbol{P}, \boldsymbol{Q}, \ldots$. We write $\boldsymbol{P}, \boldsymbol{Q}$ and mean $\boldsymbol{P} \cup \boldsymbol{Q}$ (multiset union is intended here). Similarly, $[p], \boldsymbol{Q}$ means $\{[p]\}, \boldsymbol{Q}$. We write $N \in [p]$ iff N occurs in $[p]$.

Now let $[p]$ be a branch $[N_0 \cdot N_1 \cdot \ldots \cdot N_n]$. Any contiguous subsequence of $[p]$ (possibly $[p]$ itself) is called a *partial branch* (through $[p]$). A partial branch $[q] = [N_0 \cdot N_1 \cdot \ldots \cdot N_k]$ through $[p]$ with $k \leqslant n$ is also called a *prefix of* $[p]$, written as $[q] \leqslant [p]$ or $[p] \geqslant [q]$. The node $N_k \in [p]$ is a *successor node* of $N_l \in [p]$ iff $k > l$; equivalently, we say that N_l is an *ancestor node* of N_k. The concatenation of partial branches $[p]$ and $[q]$ is denoted by $[p \cdot q]$; similarly, $[p \cdot N]$ means the extension of $[p]$ by the node N. We find it convenient to confuse a node with its label and write, for instance $[p \cdot L]$, where L is a literal, instead of $[p \cdot N]$, where $\lambda(N) = L$; the meaning of $L \in [p]$ is obtained in the same way; also, we say node L instead of the node labelled with L.

In order to memorize the fact that a branch contains a contradiction, we allow to label a branch with a "$\star$" as *closed*; we insist that if a branch is labelled as *closed* then its leaf is complementary to some of its ancestor nodes. Branches which are not labelled

as closed are said to be *open*. A literal tree is *closed* if each of its branches is closed, otherwise it is *open*.

Equality on branch sets is defined with respect to the labels and the "closed" status. More precisely, define for branches $[p]\star =_{\lambda,\lambda'} [p']\star$ iff $[p] =_{\lambda,\lambda'} [p']$ where $[N_0 \cdot \ldots \cdot N_n] =_{\lambda,\lambda'} [N'_0 \cdot \ldots \cdot N'_n]$ iff $\langle \lambda(N_0), \ldots, \lambda(N_n)\rangle = \langle \lambda'(N'_0), \ldots, \lambda'(N'_n)\rangle$. For branch sets we define $\langle \boldsymbol{P}_t, \lambda\rangle = \langle \boldsymbol{P}_{t'}, \lambda'\rangle$ iff $\boldsymbol{P}_t \approx_{\lambda,\lambda'} \boldsymbol{P}_{t'}$, where $\approx_{\lambda,\lambda'}$ is the usual multiset extension of $=_{\lambda,\lambda'}$. Finally, we say that branch set $\langle \boldsymbol{P}_t, \lambda\rangle$ is *more general* than branch set $\langle \boldsymbol{P}_{t'}, \lambda'\rangle$ iff $\langle \boldsymbol{P}_t, \lambda\rangle\delta = \langle \boldsymbol{P}_{t'}, \lambda'\rangle$, for some substitution δ.

By the previous definition literal trees are introduced as static objects. We wish to construct such literal trees in a systematic way. This is accomplished by, for instance, the restart model elimination calculus.

Definition 3 (*Branch extension, connection*). The *extension of a branch* $[p]$ *with clause* C, written as $[p] \circ C$, is the branch set $\{[p \cdot L] \mid L \in C\}$. Equivalently, in tree view this operation extends the branch $[p]$ by $|C|$ new nodes which are labelled with the literals from C. A pair of literals (K, L) is a *connection with MGU* σ iff σ is a most general unifier for K and $\overline{L}$.

Definition 4 (*Restart model elimination*). Given a clause set S in *Goal* normal form. The inference rules *extension step*, *reduction step* and *restart step* on branch sets are defined as in Fig. 2. The branch $[p]$ is called *selected branch* in all three inference rules. A restart step followed immediately by an extension step is also called a *restart extension step*.

Note that like in the usual tableaux model elimination calculus (cf. Section 1.1), an extension or a reduction step can be applicable to a negative leaf. To a positive leaf, a reduction step or a restart step can be applicable, but never an extension step. We need one more definition before turning towards derivations:

Definition 5 (*Computation rule*). A *computation rule* is a total function c which maps a tableau to one of its open branches. It is required that a computation rule is *stable under lifting*, which means that for any substitution σ, whenever $c(\boldsymbol{Q}\sigma) = [q]\sigma$ then $c(\boldsymbol{Q}) = [q]$.

The role of a computation rule is to determine in a derivation the selected branch for the next inference step:

Definition 6 (*Derivation*). Let S be a clause set in *Goal* normal form and c be a computation rule. A *restart model elimination derivation* (*RME derivation*) *of branch set* $\boldsymbol{P}_n$ *with substitution* $\sigma_1 \ldots \sigma_n$ *via* c *from* S consists of a sequence $(([\neg Goal] \equiv \boldsymbol{P}_0), \boldsymbol{P}_1, \ldots, \boldsymbol{P}_n)$ of branch sets, where for $i = 1, \ldots, n$:

(i) $\boldsymbol{P}_i$ is obtained from $\boldsymbol{P}_{i-1}$ by means of an extension step with an appropriate variant C of some clause from S and MGU σ_i, or

(ii) $\boldsymbol{P}_i$ is obtained from $\boldsymbol{P}_{i-1}$ by means of a reduction step and MGU σ_i, or

(iii) $\boldsymbol{P}_i$ is obtained from $\boldsymbol{P}_{i-1}$ by means of a restart step.

Restart model elimination (RME)

Extension step:

$$\frac{[p], \boldsymbol{P} \quad A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n}{([p \cdot A_i]\star, [p] \circ (A_1 \vee \cdots \vee A_{i-1} \vee A_{i+1} \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n), \boldsymbol{P})\sigma}$$

if

(i) $A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n$ (with $m \geqslant 1$, $n \geqslant 0$ and $i \in \{1, \ldots, m\}$) is a new variant (called *extending clause*) of a clause in S, and

(ii) $(\mathit{Leaf}([p]), A_i)$ is a connection with MGU σ. In this context A_i is called the *extension literal*.

Reduction step:

$$\frac{[p], \boldsymbol{P}}{([p]\star, \boldsymbol{P})\sigma}$$

if $(L, \mathit{Leaf}([p]))$ is a connection with MGU σ, for some node $L \in [p]$.

Restart step:

$$\frac{[p], \boldsymbol{P}}{[p] \circ \mathit{First}([p]), \boldsymbol{P}}$$

if $\mathit{Leaf}([p])$ is a positive literal.

Fig. 2. Inference rules for RME.

In each case the selected branch of the inference is determined by c. Quite often we will omit the term "via c" and mean that c is some arbitrary, given computation rule.

Finally, an *RME refutation* is an RME derivation such that $\boldsymbol{P}_n$ is closed. The term "RME" is dropped if context allows.

Notice that due to the construction of the inference rules, $\boldsymbol{P}_1$ is obtained from $\boldsymbol{P}_0$ by an extension step with some clause $\mathit{Goal} \leftarrow B_1, \ldots, B_n \in S$ and with empty substitution. This clause is called the *goal clause* of the derivation.

Note that in extension steps we can connect only with the head literals of input clauses. Since in general this restriction is too strong, because it destroys completeness, we have to "restart" the computation with a fresh copy of a negative clause. This is achieved by the restart rule, because refutations of clause sets in goal normal form always start with $\mathit{First}([p]) \equiv \neg \mathit{Goal}$, and thus only extension steps are possible to $\neg \mathit{Goal}$, which in turn introduce a new copy of a negative clause (cf. Fig. 1, right side).

1.3. Refinement: head selection function

We have argued that the RME calculus can be used as a basis for logic programming with clauses which contain disjunctions in their head. By disallowing extension steps at positive leaf literals we assure that program clauses can only be used for an extension

step such that one of the head literals is part of the connection. Hence our calculus supports the distinction into head and body literals: only head literals are used for "calling a program clause". This is one important step towards a procedural reading of disjunctive program clauses. We now go one step further, by introducing a *head selection function*. This is a means to distinguish one single head literal, which is then the only one allowed to be used for an extension step. In Example 1, in the RME refutation from Fig. 1 we used the clause $P \vee Q \leftarrow$ for an extension step at a leaf literal $\neg Q$. Now assume that we had a head selection function that returns the literal P. Then this extension step would be impossible. This is a severe restriction of the calculus; we will show later that we do not loose completeness.

Definition 7 (*Head selection function*). A *head selection function* f is a function that maps a clause $A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m$ with $n \geqslant 1$ to an atom $L \in \{A_1, \ldots, A_n\}$. L is called the *selected literal* of that clause by f. The head selection function f is required to be *stable under lifting* which means that if f selects $L\gamma$ in the instance of the clause $(A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m)\gamma$ (for some substitution γ) then f selects L in $A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m$.

Note that this head selection function has nothing to do with the selection function from SLD resolution which selects subgoals. The latter corresponds in our setting to the computation rule from Definition 5.

Definition 8 (*Derivation with head selection function*). Let f be a head selection function. An RME derivation is called a *derivation with head selection function* f if it is an RME derivation such that in every extension step only the selected literal A_i from an input clause $A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m$ is used for a connection $(Leaf([p]), A_i)$.

A head selection function obviously allows to distinguish one single head literal to be used as the only entry point during the whole derivation. Hence we arrive at a calculus which does not need contrapositives at all. From the restart property we know that a contrapositive which has a negative clause in its head can be discarded. In a derivation with head selection function we know that only the selected single positive literal has to be used. Our small derivation from the right side of Fig. 1 for Example 1 uses the clause $P \vee Q \leftarrow$ two times for an extension step. In both cases Q was used for the connection; hence if we assume a head selection function which gives Q as a selected literal, this tableau is a refutation with head selection function.

1.4. Refinement: strictness

RME with head selection function allows a procedural reading of a single program clause. Extension steps use a clause $A_1 \vee \cdots \vee A_i \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m$ as a procedure via the head A_i which is determined by the head selection function. Unfortunately, there is still a problem with the procedural interpretation, namely to explain how the remaining head literals are treated throughout a derivation. There are two possibilities to derive furtherly from a positive leaf literal: either the branch can be closed by performing a

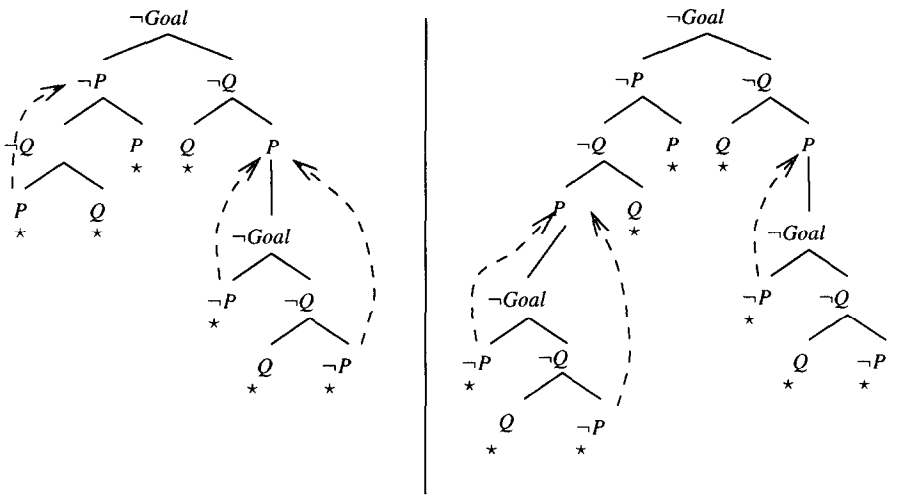

Fig. 3. RME (left side) versus strict RME (right side).

reduction step or a restart has to be done. In the right side of Fig. 1 both possibilities are contained. The literal P is used in a reduction step, hence it is the leaf of the leftmost closed branch, and the rightmost path containing the P was extended by a restart step. In strict RME derivations we forbid reduction steps at positive leaf literals.

Definition 9 (*Strict RME*). An RME derivation is called a *strict RME derivation* iff in the connection $(L, Leaf([p]))$ of every reduction step, $Leaf([p])$ is a negative literal.

In Fig. 3 we oppose our sample RME refutation to a strict RME refutation. Note that in a strict RME derivation, reduction steps still can be contained. However, they are only of one single kind, namely when a negative leaf literal is used for a connection with a positive literal within the branch.

For a procedural interpretation of program clauses and its role in a refutation, we now can assume for strict RME refutations with head selection function, that all reduction steps close branches with a negative leaf literal. These reduction steps are interpreted very naturally by the following view to the restart concept. Let $A_1 \vee A_2 \leftarrow B$ be a program clause and A_1 be the selected literal. Then a call to this clause can be done via A_1 within a refutation and, in addition, one has to prove the original goal with the extra assumption A_2. Instead of adding the fact A_2 to the clause set for this additional proof strict RME allows the closing of a path by reduction steps to the path literal A_2 which obviously has the same effect.

1.5. Refinement: regularity

The *regularity check* for model elimination says that it is never necessary to construct a tableau where a literal occurs twice (or even more often) along a branch. Expressed operationally, it says that it is never necessary to repeat a previously derived subgoal

(viewing open leaves as subgoals). For a semantic interpretation take the view that a branch constitutes a partial interpretation, and any clause containing a literal from the branch would be satisfied by this interpretation. Hence this clause need not be considered for "eliminating" the interpretation given by the branch. From a logic programming perspective this can be seen as a very simple loop check.

Regularity is easy to implement, at least approximately, and it is one of the more effective restrictions for model elimination procedures. Unfortunately, the regularity check is *not* compatible with RME. This is rather easy to see since after a restart step it might be necessary to repeat—in parts—a refutation derived so far up to the restart step. However, what can be achieved is *blockwise regularity*.

Definition 10 (*Blockwise regularity*). Let $[p]$ be a branch written as follows, and the A and B are atoms:

$$[p] = [\neg B_1^1 \cdot \ldots \cdot \neg B_{k_1}^1 A^1 \neg B_1^2 \cdot \ldots \cdot \neg B_{k_2}^2 A^2 \cdot \ldots \cdot A^{n-1} \neg B_1^n \cdot \ldots \cdot \neg B_{k_n}^n].$$

Then $[p]$ is called *blockwise regular* iff

(i) $A^i \neq A^j$ for $1 \leqslant i, j \leqslant n-1$, $i \neq j$ (*regularity with respect to positive literals*), and

(ii) $B_i^l \neq B_j^l$ for $1 \leqslant l \leqslant n$, $1 \leqslant i, j \leqslant k_l$, $i \neq j$ (*regularity inside blocks*).

A branch set is called *blockwise regular* iff every branch in it is blockwise regular. Similarly, a derivation is called *blockwise regular* iff each of its branch sets is blockwise regular.

For example, both derivations in Fig. 3 are blockwise regular.

1.6. Completeness

In this section we give a ground completeness result for the weakest of the above introduced calculi, namely strict RME with head selection function and blockwise regularity. As a trivial corollary one gets ground completeness of all other introduced variants of RME. The following technical lemma is used for the completeness proof.

Lemma 11. *Let S be a minimal unsatisfiable ground clause set with $L_1 \vee \cdots \vee L_n \in S$ $(n \geqslant 1)$. Then for every subclause $C \subset L_1 \vee \cdots \vee L_n$ there is a set*

$$S_C \subseteq (S \setminus \{L_1 \vee \cdots \vee L_n\})$$

such that $S_C \cup \{C\}$ is minimal unsatisfiable.

Proof. It is clear that $(S \setminus \{L_1 \vee \cdots \vee L_n\}) \cup \{C\}$ is unsatisfiable, because otherwise a model for this set would constitute a model for S itself. Let $S'_C \subseteq (S \setminus \{L_1 \vee \cdots \vee L_n\}) \cup \{C\}$ be any minimal unsatisfiable subset. It suffices to show that $C \in S'_C$, because then $S_C := S'_C \setminus \{C\}$ proves the lemma.

Hence suppose, to the contrary, that $C \notin S'_C$. But then $S'_C \subset S$, and the strict inclusion contradicts the given fact that S is *minimal* unsatisfiable. □

Theorem 12 (Ground completeness). *Let S be a minimal unsatisfiable ground clause set in Goal normal form, c be a computation rule and f be a head selection function. Then there is a blockwise regular, strict RME refutation via c and f.*

Proof. We will show the existence of the claimed refutation for *some* computation rule. Then, with a suitable switching lemma *à la* [17] this refutation can be reordered according to c. Since this is straightforward for the *ground* case considered here, we will omit the switching lemma and its proof.

Let $k(S)$ denote the number of occurrences of positive literals in S minus the number of nonnegative clauses[3] in S ($k(S)$ is a measure for the "Horn-ness" of S; it is related to the well-known *excess literal parameter*). Now we prove the claim by induction on $k(S)$.

Base case: $k(S) = 0$. By well-known completeness results for ME (see e.g. [1, 15, 20]) there exists a regular ME refutation of S. It is also well known that ME is complete when restricted to negative goal clauses. Since the only negative clause in S is $\neg Goal$, it must also be the goal clause of this refutation. This refutation also trivially is strict.

Induction step: $k(S) > 0$. As the induction hypothesis assume the result to hold for clause sets S' satisfying the preconditions and $k(S') < k(S)$.

Since $k(S) > 0$ there is in S a disjunctive clause

$$C = A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n \quad (m \geqslant 2, n \geqslant 0).$$

Suppose that the head selection function f selects A_i in C. By Lemma 11 there is a set

$$S' \subseteq ((S \setminus \{A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n\}) \cup \{A_i \leftarrow B_1, \ldots, B_n\})$$

such that S' is minimal unsatisfiable. S' still is in *Goal* normal form. Hence, by the induction hypothesis there is a closed, blockwise regular, and strict RME refutation D' of S' and head selection function f', where f' is the same as f, but selects in $A_i \leftarrow B_1, \ldots, B_n$ the head A_i (there is no other choice for f'). Let $\boldsymbol{P}'$ be the closed tableau derived by D'. Now replace every extension step in D' using the extending clause $A_i \leftarrow B_1, \ldots, B_n$ by $A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$. This leaves us with a derivation from S and f, resulting in an open tableau whose open branches all end in a literal from $\{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m\}$.

Now we delete from this tableau all subtrees below all positive inner nodes that do not have any positive literals as ancestors. More formally, replace every branch of the form $[p \cdot A' \cdot q]\star$ by $[p \cdot A']$, where $[p]$ consists of negative nodes only, A' is a positive node, and $[q]$ is non-empty. Let $\boldsymbol{P}''$ be the resulting open tableau. $\boldsymbol{P}''$ can be thought of to be constructed in an RME derivation D'' of S and f which does not further extend at the first positive nodes in each branch.

Every open branch in $\boldsymbol{P}''$ now takes the form $[p \cdot A']$, for some $[p]$ consisting of negative nodes only, and A' is a positive node, stemming from some disjunctive clause from S.

We will show how D'' can be extended such that each branch $[p \cdot A']$ is extended to a closed subtree below it. It suffices to show the construction for one of these branches,

[3] A *nonnegative clause* is a clause containing at least one positive literal.

because the construction can be applied to each of them, one after the other, which eventually leads to the desired refutation.

Hence let $[p \cdot A']$ be one of those branches. The leaf literal A' stems from a clause

$$A'_1, \ldots, A'_{m'} \leftarrow B'_1, \ldots, B'_{n'} \in S \quad (m' \geqslant 2, n' \geqslant 0),$$

where $A' \equiv A'_j$ for some $j \in \{1, \ldots, m'\}$. Since S is given as minimal unsatisfiable, we can find by Lemma 11 a set

$$S'' \subseteq S \setminus \{A'_1, \ldots, A'_{m'} \leftarrow B'_1, \ldots, B'_{n'}\}$$

such that $S'' \cup \{A' \leftarrow\}$ is minimal unsatisfiable. Clearly, $k(S'' \cup \{A' \leftarrow\}) < k(S)$ (we replace a disjunctive clause by definite unit clause and possibly delete clauses). Since $S'' \cup \{A' \leftarrow\}$ still is *Goal* normal form, we can find by the induction hypothesis a closed, blockwise regular and strict RME refutation D''' of $S'' \cup \{A' \leftarrow\}$ and f'', where f'' is the same as f, except that it selects A' in $A' \leftarrow$ (there is no other choice).

Next append D'' with a restart step applied to $[p \cdot A']$. This results in a branch $[p \cdot A' \cdot \neg Goal]$. Then further append D''' to this derivation, however each branch $[\neg Goal \cdot q]$ in D''' being replaced by $[p \cdot A' \cdot \neg Goal \cdot q]$. This leaves us with a derivation D'''' from $S \cup \{A' \leftarrow\}$ with head selection function f'' and such that a closed subtree below $[p \cdot A']$ comes up.

Now we have to turn D'''' into a derivation from S alone. By construction, the clause $A' \leftarrow$ is used in D'''' for extension steps only in the subtree below $[p \cdot A']$, and always results in a closed branch of the form $[p \cdot A' \cdot \neg Goal \cdot q' \cdot \neg A' \cdot A']\star$. Hence we can replace each of these extension steps by a reduction step, yielding a closed branch $[p \cdot A' \cdot \neg Goal \cdot q' \cdot \neg A']\star$ instead. This gives us a strict derivation D from S alone using the head selection function f.

It remains to show that this construction preserves blockwise regularity. The regularity inside blocks trivially carries over from D''' to D''''. For the regularity with respect to positive literals, the critical case is the node A' which is with respect to positive nodes the only difference between the tableau derived in D''' and D''''. However, recall that D''' is a refutation of $S'' \cup \{A' \leftarrow\}$, which is *minimal* unsatisfiable. This implies that S'' cannot contain a disjunctive clause with a head literal A'. But then, the literal A' can occur in the tableau derived in D''' *only* at a leaf position, stemming from extension steps with $A' \leftarrow$. These usages, however, were eliminated in the transition from D'''' to D'. Hence D is blockwise regular. □

2. Computing answers

We now refine the RME calculus by one more concept, namely *answer computation*. By this, we mean the possibility to compute most general solutions to problem statements in the sense made popular for Horn clause logic programming.

In order to do so within the RME calculus, we have to presuppose a certain structure of the input clause set. More specifically, we assume as given a satisfiable clause set P, together with one single *query*, which is a clause of the form $\leftarrow G_1 \wedge \cdots \wedge G_n$, where

the G are positive literals. Notice that we do *not* require P to be a program (i.e., a finite set of program clauses, cf. Section 1.2). While such a restriction would be unnecessary for the results of this section, it will be an issue again in Section 3 (cf. Note 1). The restriction to one single query clause is a bit artificial, but supports the clarity of presentation. The generalization to more than one query clause is straightforward.

We will often abbreviate a query as $\leftarrow Q$, where Q stands for the conjunction of Gs. The clause set S is the transformation of $P \cup \{\leftarrow Q\}$ into goal normal form.

Definition 13 (*Answers*). If $\leftarrow Q$ is a query, and $\theta_1, \ldots, \theta_m$ are substitutions for the variables from Q, then $Q\theta_1 \vee \cdots \vee Q\theta_m$ is an *answer* (for S). An answer $Q\theta_1 \vee \cdots \vee Q\theta_m$ is a *correct answer* if $P \models \forall(Q\theta_1 \vee \cdots \vee Q\theta_m)$. Now let an RME refutation of S with top clause $\leftarrow Goal$ and substitution σ be given. Assume that this refutation contains m extension steps with the query, i.e., it contains m times an extension step with the clause $Goal \leftarrow Q\rho_i$, where ρ_i is the renaming substitution of this step. Let $\sigma_i = \rho_i\sigma|_{dom(\rho_i)}$. Then $Q\sigma_1 \vee \cdots \vee Q\sigma_m$ is a *computed answer* (for S).

That is, we simply collect applications of the query clause to obtain the answer. This idea is, of course, not new. For resolution, question answering was invented in the early paper [13]; the idea is to attach answer literals to trace the usages of the query in the resolution proof (see also [8] and Section 5.3 below).

Example 14. As an example take the following program P and query $\leftarrow Q$ (constants and function symbols start with lower case letters and variables with upper case letters):

$$\begin{array}{lll} P: & P(X) \leftarrow Q(X), Q(X) \\ & Q(a), Q(b) \leftarrow \\ \leftarrow Q: & \leftarrow P(X). \end{array}$$

A refutation together with a computed answer is given in Fig. 4.

Before we prove answer-completeness we explicitly give a lifting theorem for restart model elimination. The first part of the theorem can be regarded as a proof of *refutational completeness* (because equality among branch sets takes the "closed" status into account, cf. Definition 2), i.e., if $\boldsymbol{P}_n$ is closed, then $\boldsymbol{P}'_n$ is closed as well. The second part will be used in the proof of *answer-completeness* (Theorem 16).

2.1. Lifting

Theorem 15 (Lifting theorem for restart model elimination). *Let S' be a set of ground instances of clauses taken from a clause set S in Goal normal form. Assume there exists a blockwise regular RME derivation $D' \equiv (\boldsymbol{P}'_0, \boldsymbol{P}'_1, \ldots, \boldsymbol{P}'_n)$ from S' with goal clause $C'_0 \in S'$. Then there exists a blockwise regular RME derivation $D \equiv (\boldsymbol{P}_0, \boldsymbol{P}_1, \ldots, \boldsymbol{P}_n)$ from S with some goal clause $C_0 \in S$ and substitution δ such that $\boldsymbol{P}_n\delta = \boldsymbol{P}'_n$.*

Furthermore, there exists a substitution δ such that $\boldsymbol{P}'_i$ is obtained from $\boldsymbol{P}'_{i-1}$ by an extension step with clause $C' \in S'$ if and only if $\boldsymbol{P}_i$ is obtained from $\boldsymbol{P}_{i-1}$ by an

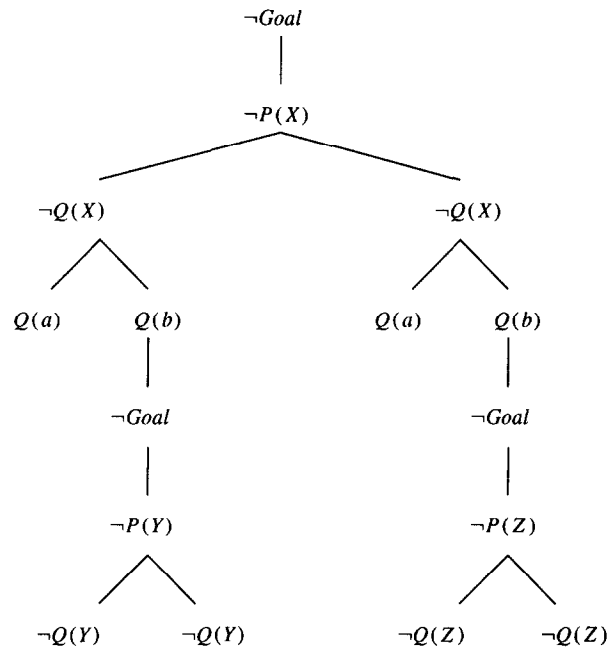

Fig. 4. A refutation of the *Goal* normal form $P \cup \{\leftarrow Goal, Goal \leftarrow P(X)\}$ of the clauses in Example 14, depicted as a tree; the computed answer is $P(a) \vee P(b) \vee P(b)$ where the substitution $\{X \leftarrow a, Y \leftarrow b, Z \leftarrow b\}$ is applied.

extension step with a clause $C \in S$ *such that* $C\rho_i\sigma\delta = C'$, *where* ρ_i *is the renaming substitution applied in that extension step.*

Proof. The basic proof plan is to show by induction on n that $\boldsymbol{P}_n$ can be mapped by application of a certain substitution δ_n to $\boldsymbol{P}'_n$. This approach would suffice to lift a derivation to the first-order level. However, as stated in the second part of the theorem, we need moreover a lifting result for the clauses used in extension steps.

In order to make things technically manageable we first define a clause set S_v as follows: S_v is a set of, say l, pairwise variable disjoint clauses, and S_v contains for every ground instance $C_k\gamma_k \in S'$ $(k = 1, \ldots, l)$ of a clause $C_k \in S$ a variant $C_k\tau_k$. Furthermore $C_k\tau_k$ is supposed to be variable disjoint from S.

At first we will show that there exists one single ground substitution γ which can be used instead of the individual γ_k. More precisely, we define first $\gamma'_k = \tau_k^{-1}\gamma_k|_{vcod(\tau_k)}$, because then we have

$$(C_k\tau_k)\gamma'_k = C_k\gamma_k. \tag{1}$$

Moreover, since the clauses in S_v are assumed to be pairwise variable disjoint (by means of the τ_k) and because of the domain restriction of the γ'_k it follows $(C_k\tau_k)\gamma'_k = (C_k\tau_k)\gamma'_1 \ldots \gamma'_l$. But then, defining $\gamma = \gamma'_1 \ldots \gamma'_l$ and using (1) we recognize more generally that $S_v\gamma = S'$.

It follows with D' being a derivation from S' that D' is also a derivation from $S_v\gamma$. We will show how to lift D' from $S_v\gamma$ to the first-order level. In order to do so we have to define a slightly more general induction invariant $I(n)$ than the theorem gives us. In present notation it reads as follows:

$I(n)$: there exists an RME derivation $(\boldsymbol{P}_0, \boldsymbol{P}_1, \ldots, \boldsymbol{P}_n)$ from S with substitution $\sigma_1 \ldots \sigma_n$, and there exists a substitution δ_n such that

- *invariant* 1: $\boldsymbol{P}_n\delta_n = \boldsymbol{P}'_n$ and
- *invariant* 2: whenever $\boldsymbol{P}'_i$ $(i = 1, \ldots, n)$ is obtained from $\boldsymbol{P}'_{i-1}$ by an extension step with a clause $C_v\gamma \in S_v\gamma$, then $\boldsymbol{P}_i$ is obtained from $\boldsymbol{P}_{i-1}$ by an extension step with some clause $C \in S$ such that

$$C\rho_i\sigma_1 \ldots \sigma_n\delta_n = C_v\gamma$$

where ρ_i is the renaming substitution used in the ith step to obtain a new variant of C.

Clearly, $I(n)$ proves the theorem using the identity $C' = C_v\gamma$, and defining $\sigma := \sigma_1 \ldots \sigma_n$ and $\delta := \delta_n$. The if-direction of the theorem's statement (i.e., that D does not need more clauses for extension steps than D') follows from the construction given below.

It thus remains to prove $I(n)$ (induction on n):

$n = 0$: Trivial, because with $\boldsymbol{P}'_0 \equiv [\neg Goal]$ we can also set $\boldsymbol{P}_0 \equiv [\neg Goal]$ and define $\delta_0 = \varepsilon$ (the empty substitution); invariant 2 holds vacuously.

$(n-1) \rightarrow n$: Let $n > 0$ and suppose $I(n-1)$ to hold for the derivation $(\boldsymbol{P}'_0, \boldsymbol{P}'_1, \ldots, \boldsymbol{P}'_{n-1})$. $I(n-1)$ gives us that there exists a substitution δ'_{n-1} such that

$$\boldsymbol{P}_{n-1}\delta_{n-1} = \boldsymbol{P}'_{n-1}, \tag{2}$$

$$C\rho_i\sigma_1 \ldots \sigma_{n-1}\delta_{n-1} = C_v\gamma \tag{3}$$

under the provisos stated above in the definition of $I(n)$. In order to prove $I(n)$ we make a case analysis with respect to the inference rule applied to $\boldsymbol{P}'_{n-1}$:

Reduction step: $\boldsymbol{P}'_{n-1}$ contains a branch $[p']$ of the form $[p'] = [\ldots \cdot A \cdot \ldots \cdot \overline{A}]$ to be closed, i.e., $\boldsymbol{P}'_n$ contains $[p']\star$. From the given invariant (2) we learn that $\boldsymbol{P}_{n-1}$ in particular contains a branch $[p]$ with $[p\delta_{n-1}] = [p']$. The branch $[p]$ is of the form $[p] = [\ldots \cdot K \cdot \ldots \cdot \overline{L}]$, where $K\delta_{n-1} = A = L\delta_{n-1}$. In other words, δ_{n-1} is a unifier for K and L. Since S_v is assumed to be a set of variants completely variable disjoint from S we can also assume that γ neither acts on the variables from S nor on the variables occurring in the variants taken from S to build the derivation $(\boldsymbol{P}_0, \ldots, \boldsymbol{P}_{n-1})$. But then $\boldsymbol{P}_{n-1}\gamma = \boldsymbol{P}_{n-1}$. Application of δ_{n-1} yields

$$\boldsymbol{P}_{n-1}\gamma\delta_{n-1} = \boldsymbol{P}_{n-1}\delta_{n-1} \overset{(2)}{=} \boldsymbol{P}'_{n-1} = \boldsymbol{P}'_{n-1}\gamma\delta_{n-1}. \tag{4}$$

The last identity is trivial ($\boldsymbol{P}'_{n-1}$ is ground) and is needed below.

More specifically the following holds

$$K\gamma\delta_{n-1} = K\delta_{n-1} = A = L\delta_{n-1} = L\gamma\delta_{n-1}. \tag{5}$$

Next we turn to the clauses used in previous extension steps (cf. given invariant 2). Again, since γ does not act on the variants taken from S it also holds $C\rho_i\sigma_1 \ldots \sigma_{n-1}\gamma\delta_{n-1} = C\rho_i\sigma_1 \ldots \sigma_{n-1}\delta_{n-1}$. Thus we conclude

$$C\rho_i\sigma_1 \ldots \sigma_{n-1}\gamma\delta_{n-1} = C\rho_i\sigma_1 \ldots \sigma_{n-1}\delta_{n-1} \overset{(3)}{=} C_v\gamma = C_v\gamma\delta_{n-1}. \tag{6}$$

The last identity holds trivially because $C_v\gamma$ is ground.

Eqs. (4), (5), (6) tell us that $\gamma\delta_{n-1}$ is a simultaneous unifier for the respective leftmost and rightmost terms in these equations. Hence there also exists an MGU σ_n and a substitution δ_n such that

$$\sigma_n\delta_n = \gamma\delta_{n-1}. \tag{7}$$

By construction, σ_n is an MGU for K and L. Hence we can apply a reduction step to $[p] \in \boldsymbol{P}_{n-1}$ with MGU σ_n to obtain $\boldsymbol{P}_n = ((\boldsymbol{P}_{n-1} \setminus \{[p]\}) \cup \{[p]\star\})\sigma_n$. Altogether we conclude

$$\begin{aligned}
\boldsymbol{P}_n\delta_n &= ((\boldsymbol{P}_{n-1} \setminus \{[p]\}) \cup \{[p]\star\})\sigma_n\delta_n \\
&\overset{(7),(4)}{=} ((\boldsymbol{P}'_{n-1} \setminus \{[p']\}) \cup \{[p']\star\})\gamma\delta_{n-1} \\
&\overset{(*)}{=} ((\boldsymbol{P}'_{n-1} \setminus \{[p']\}) \cup \{[p']\star\}) = \boldsymbol{P}'_n.
\end{aligned}$$

The step $(*)$ is justified by the fact that $\boldsymbol{P}'_{n-1}$ is ground. Note that this chain just proves invariant 1; the remaining invariant 2 is shown as follows:

$$C\rho_i\sigma_1 \ldots \sigma_{n-1}\sigma_n\delta_n \overset{(7)}{=} C\rho_i\sigma_1 \ldots \sigma_{n-1}\gamma\delta_{n-1} \overset{(6)}{=} C_v\gamma.$$

Extension step: $\boldsymbol{P}'_{n-1}$ contains a branch $[p']$ of the form $[p'] = [\ldots \cdot A]$ to be extended with a clause $\overline{A} \vee R \in S_v\gamma$, i.e.,

$$\boldsymbol{P}'_n = (\boldsymbol{P}'_{n-1} \setminus \{[p']\}) \cup \{[p' \cdot \overline{A}]\star\} \cup [p'] \circ R.$$

From the given invariant (2) we learn that $\boldsymbol{P}_{n-1}$ in particular contains a branch $[p]$ with $[p\delta_{n-1}] = [p']$. The branch $[p]$ is of the form $[p] = [\ldots \cdot L]$, and thus, in particular $A = L\delta_{n-1}$.

Let $\overline{K_v} \vee Q_v \in S_v$ be the lifted version of $\overline{A} \vee R$, i.e.,

$$(\overline{K_v} \vee Q_v)\gamma = \overline{A} \vee R. \tag{8}$$

Recall from the assumption stated at the beginning, that $\overline{K_v} \vee Q_v$ is a new variant by means of some renaming substitution τ, i.e., $\overline{K_v} \vee Q_v = (\overline{K} \vee Q)\tau$ for some $\overline{K} \vee Q \in S$. Let $(\overline{K} \vee Q)\rho_n$ be a new respective variant. We will show how to carry out an extension step with that variant.

From the last equation and (8) it follows

$$\begin{aligned}
\overline{A} \vee R &= (\overline{K_v} \vee Q_v)\gamma = (\overline{K} \vee Q)\tau\gamma = ((\overline{K} \vee Q)\rho_n)\rho_{n-1}^{-1}\tau\gamma \\
&= ((\overline{K} \vee Q)\rho_n)\rho_{n-1}^{-1}\tau\gamma\delta_{n-1}.
\end{aligned} \tag{9}$$

The last identity is justified by the fact that δ_{n-1} is applied to a ground clause, and hence does not alter the clause.

The renaming substitution ρ_n introduces new variables; hence ρ_n^{-1} does not affect P_{n-1}. Also, τ does not affect P_{n-1}, because P_{n-1} is built from new variants. Together we obtain that $P_{n-1} = P_{n-1}\rho_n^{-1}\tau$. Further, as in the case of the reduction step, we can also assume that γ does neither act on the variables from S nor on the variables occurring in the variants taken from S to build the derivation $(P_0, \ldots, P_{n-1})$. But then $P_{n-1}\gamma = P_{n-1}$. Putting things together we obtain:

$$P_{n-1}\rho_n^{-1}\tau\gamma\delta_{n-1} = P_{n-1}\delta_{n-1} \overset{(2)}{=} P'_{n-1} = P'_{n-1}\rho_n^{-1}\tau\gamma\delta_{n-1}. \tag{10}$$

The last identity is trivial (P'_{n-1} is ground) and is needed below.

Recall from above that we extended P'_{n-1} at a branch $[p']$ with leaf A to obtain P'_n. From $A = L\delta_{n-1}$, as given, we can now conclude with (10) even $L\rho_n^{-1}\tau\gamma\delta_{n-1} = A$. But then we have

$$L\rho_n^{-1}\tau\gamma\delta_{n-1} = A \overset{(9)}{=} (K\rho_n)\rho_n^{-1}\tau\gamma\delta_{n-1}. \tag{11}$$

Next we turn to the clauses used in previous extension steps (cf. given invariant 2). By the same line of reasoning as for P_{n-1} above we can assume that $\rho_n^{-1}\tau$ does not affect a clause C_v whose ground instance $C_v\gamma$ is used in the ground derivation. Thus we conclude

$$C_v\gamma = C_v\rho_n^{-1}\tau\gamma = C_v\rho_n^{-1}\tau\gamma\delta_{n-1}. \tag{12}$$

The last identity holds because δ_{n-1} is applied to a ground clause.

Since the derivation from S uses new variants, and the MGUs used there can be supposed to introduce no new variables we have

$$C\rho_i\sigma_1 \ldots \sigma_{n-1} = C\rho_i\sigma_1 \ldots \sigma_{n-1}\rho_n^{-1}\tau.$$

Again, since γ does not act on the variants taken from S, $C\rho_i\sigma_1 \ldots \sigma_{n-1}\gamma = C\rho_i\sigma_1 \ldots \sigma_{n-1}$ also holds. Using these identities and applying δ_{n-1} we conclude

$$\begin{aligned} C\rho_i\sigma_1 \ldots \sigma_{n-1}\rho_n^{-1}\tau\gamma\delta_{n-1} &= C\rho_i\sigma_1 \ldots \sigma_{n-1}\delta_{n-1} \\ &\overset{(3)}{=} C_v\gamma \overset{(12)}{=} C_v\rho_n^{-1}\tau\gamma\delta_{n-1}. \end{aligned} \tag{13}$$

Eqs. (10), (11), (13) tell us that $\rho_n^{-1}\tau\gamma\delta_{n-1}$ is a simultaneous unifier for the respective leftmost and rightmost terms in these equations. Hence there also exists an MGU σ_n and a substitution δ_n such that

$$\sigma_n\delta_n = \rho_n^{-1}\tau\gamma\delta_{n-1}. \tag{14}$$

By (11) and (14), σ_n is an MGU for L and $K\rho_n$. Hence we can apply an extension step to $[p] \in P_{n-1}$ with the variant clause $(\overline{K} \vee Q)\rho_n$ and MGU σ_n to obtain

$$P_n = ((P_{n-1} \setminus [p]) \cup \{[p \cdot (\overline{K}\rho_n)]\star\} \cup [p] \circ Q\rho_n)\sigma_n.$$

Altogether we conclude

$$\begin{aligned}
\boldsymbol{P}_n\delta_n &= ((\boldsymbol{P}_{n-1} \setminus [p]) \cup \{[p \cdot (\overline{K}\rho_n)]\star\} \cup [p] \circ Q\rho_n)\sigma_n\delta_n \\
&\overset{(14),(10)}{=} ((\boldsymbol{P}'_{n-1} \setminus [p']) \cup \{[p' \cdot (\overline{K}\rho_n)]\star\} \cup [p'] \circ Q\rho_n)\sigma_n\delta_n \\
&\overset{(14),(9)}{=} ((\boldsymbol{P}'_{n-1} \setminus [p']) \cup \{[p' \cdot \overline{A}]\star\} \cup [p'] \circ R) \\
&= \boldsymbol{P}'_n.
\end{aligned}$$

Note that this chain of reasoning just proves invariant 1.

We still have to prove invariant 2: first, invariant 2 has to be proved for all clauses used up to, but not including this extension step:

$$C\rho_i\sigma_1 \ldots \sigma_{n-1}\sigma_n\delta_n \overset{(14)}{=} C\rho_i\sigma_1 \ldots \sigma_{n-1}\rho_n^{-1}\tau\gamma\delta_{n-1} \overset{(13)}{=} C_v\gamma.$$

Second, invariant 2 has to be proved for the clause used in this extension step. For this note that ρ_n renames $\overline{K} \vee Q$ to a new variant. Hence $((\overline{K} \vee Q)\rho_n)\sigma_1 \ldots \sigma_{n-1} = (\overline{K} \vee Q)\rho_n$. From this it follows

$$\begin{aligned}
((\overline{K} \vee Q)\rho_n)\sigma_1 \ldots \sigma_{n-1}\sigma_n\delta_n &= (\overline{K} \vee Q)\rho_n)\sigma_n\delta_n \\
&\overset{(14)}{=} ((\overline{K} \vee Q)\rho_n)\rho_n^{-1}\tau\gamma\delta_{n-1} \\
&\overset{(9)}{=} (\overline{K_v} \vee Q_v)\gamma.
\end{aligned}$$

Now invariants 1 and 2 have been shown, which concludes the proof of the extension step.

Restart step: Since in a restart step no substitution is applied we can take $\delta_n := \delta_{n-1}$. Together with the induction hypothesis the result follows trivially.

To see the blockwise regularity of the lifted derivation D, assume to the contrary of the theorem, that the lifted version D' is not blockwise regular. Then some branch $[p]$ along the refutation D' is not globally regular. This means that one of the inequality constraints stated in the definition of blockwise regularity is violated. Thus $[p]$ contains two occurrences of a literal B which violate one of these constraints. However, with the branch $[p]$ being a lifted version of a branch $[p']$ in D' the inequality constraint must be violated in $[p']$ as well. This plainly contradicts the assumption that D' is blockwise regular. Hence D' is also blockwise regular. □

2.2. *Answer-completeness*

Theorem 16 (Answer-completeness of RME). *Let P be a satisfiable clause set, $\leftarrow Q$ be a query and $Q\theta_1 \vee \cdots \vee Q\theta_l$ be a correct answer for P. Then there exists a blockwise regular, strict RME refutation with head selection function from the Goal normal form S of $P \cup \{\leftarrow Q\}$, with computed answer $Q\sigma_1 \vee \cdots \vee Q\sigma_m$ such that $Q\sigma_1 \vee \cdots \vee Q\sigma_m$ entails $Q\theta_1 \vee \cdots \vee Q\theta_l$, i.e.,*

$$\exists\delta\, \forall i \in \{1, \ldots, m\}\, \exists j \in \{1, \ldots, l\}\, Q\sigma_i\delta = Q\theta_j.$$

Informally, the theorem states that for every given correct answer we can find a computed answer which can be instantiated by means of a *single* substitution δ to a

subclause of the given answer, and hence implies it. To obtain this result we have to demand *one single* substitution δ which maps any of the clauses $C\rho_i\sigma$ used in extension steps to the respective clause on the ground level.

Clearly, this result is harder to establish and more relevant than a lifting result for SLI resolution in [18] which "moves the $\exists$-quantification inside": in our words, they state that for every application of an input clause at the ground level there exists an application at the first-order level, and there exists a substitution to map *this instance* to the ground level. As Example 17 shows, the approach of [18] cannot handle the case correctly if there are variable interdependencies in disjunctive answers.

Example 17. To see this, let us consider the clause set $P = \{P(f(X)) \leftarrow\}$ and the query $\leftarrow P(Z)$. Then according to [18], the singleton $A = \{P(Z) \vee P(f(Z))\}$ is a correct and complete answer set, although A does not subsume $P(f(X))$. Thus we would be incomplete. On the other hand, if we permit each literal to be substituted separately, in order to fix this problem, then we would be incorrect. For this let us consider the clause set $P' = \{P(X) \vee P(f(X)) \leftarrow\}$. Then we could get also the answer A with respect to the query $\leftarrow P(Z)$ from above. But we could factor it to $P(f(Z))$ which is not entailed by P'.

Unfortunately, we can *not* obtain a result stating that the computed answer contains less (or an equal number of) literals than the given answer. This behaviour sometimes results in confusing answers. For instance, take again the clause set from Example 14:

$$\begin{aligned} P: \quad & P(X) \leftarrow Q(X), Q(X) \\ & Q(a), Q(b) \leftarrow \\ \leftarrow Q: \quad & \leftarrow P(X). \end{aligned}$$

The refutation from Fig. 4 computes the answer $P(a) \vee P(b) \vee P(b)$. Although $P(a) \vee P(b)$ is a correct answer, RME will not compute it. The reason for this is that two identical instances $\leftarrow P(b)$ of the query have to be used. Now follow the proof of Theorem 16.

Proof. Given the correct answer $Q\theta_1 \vee \cdots \vee Q\theta_l$ we know by definition $P \models \forall(Q\theta_1 \vee \cdots \vee Q\theta_l)$. Hence we conclude that $P \cup \{\neg\forall(Q\theta_1 \vee \cdots \vee Q\theta_l)\}$ is unsatisfiable. By transforming this into CNF we get the unsatisfiable set of clauses $S' = P \cup \{\neg Q\theta_1\tau_1, \ldots, \neg Q\theta_l\tau_l\}$ where each τ_i $(1 \leqslant i \leqslant l)$ substitutes new Skolem constants for the free variables of $Q\theta_i$.

With the abbreviation $\theta_i' = \theta_i\tau_i$ for all $1 \leqslant i \leqslant l$, we get an unsatisfiable set of clauses

$$S' = P \cup \{\neg Q\theta_1', \ldots, \neg Q\theta_l'\}.$$

By the Herbrand–Löwenheim–Skolem theorem there exists an unsatisfiable ground clause set

$$S'' = P' \cup \{\neg Q\theta_1', \ldots, \neg Q\theta_l'\}$$

where P' is a finite set of ground instances of clauses from P. From S'' we select a *minimal* unsatisfiable subset

$$S''' = P'' \cup \{\neg Q\theta'_1, \ldots, \neg Q\theta_r\}$$

where $P'' \subseteq P'$, and (without loss of generality) $r \leqslant l$. From ground completeness of RME (Theorem 12) we learn that there exists an RME refutation of the *Goal* normal form of S''', i.e., there exists an RME refutation D' of

$$S'''' = P''_{Goal} \cup \{Goal \leftarrow Q\theta'_1, \ldots, Goal \leftarrow Q\theta'_r\} \cup \{\leftarrow Goal\}.$$

Here, a clause set P_{Goal} is obtained from a clause set P by replacing every purely negative clause $\neg B_1 \vee \cdots \vee \neg B_n$ by $Goal \vee \neg B_1 \vee \ldots \vee \neg B_n$. The minimality condition ensures that each of the clauses $\{Goal \leftarrow Q\theta'_1, \ldots, Goal \leftarrow Q\theta'_r\}$ in S'''' is used at least once for an extension step. Let $m \geqslant r$ be the total number of extension steps carried out with clauses from that set.

D' is a refutation, i.e., a derivation of a closed branch set $\boldsymbol{P}_n$. By the lifting theorem there exists a blockwise regular RME derivation D of some more general branch set $\boldsymbol{P}_n$ from $\boldsymbol{P}_{Goal} \cup \{Goal \leftarrow Q\} \cup \{\leftarrow Goal\}$ since the second part of the lifting theorem assures that even single applications of extension steps are lifted, we conclude that the structure of the lifted tableau is invariant, i.e., strictness and head selection property lifts as well. Since only the empty branch set is more general than itself we conclude that $\boldsymbol{P}_n$ is also a refutation. This proves refutational completeness.

Next we turn to answer-completeness, proper. The second part of the lifting theorem gives us that for any extension step in D' with clause $Goal \leftarrow Q\theta'_{f(i)}$ (where f is a surjection from $\{1, \ldots, m\}$ onto $\{1, \ldots, r\}$) there is exactly one extension step in D with the clause $\{Goal \leftarrow Q\rho_i\}$ $(i \in \{1, \ldots, m\})$, where ρ_i is the renaming substitution of that step. Furthermore (by the lifting theorem) there is a substitution δ' such that

$$Goal \leftarrow Q\rho_i\sigma\delta' = Goal \leftarrow Q\theta'_{f(i)}. \tag{15}$$

This, however, is equivalent to the condition that every element in the disjunction

$$Ans = Q\sigma_1 \vee \cdots \vee Q\sigma_r \vee Q\sigma_{r+1} \vee \cdots \vee Q\sigma_m$$

where $\sigma_i = \rho_i\sigma|_{dom(\rho_i)}$ (for $i = 1, \ldots, m$), is mapped by application of δ' into some element of $Q\theta'_1 \vee \cdots \vee Q\theta'_r$. Note here that *Ans* is nothing but the computed answer substitution.

Recall that $Q\theta'_k = Q\theta_k\tau_k$ (for $k = 1, \ldots, r$), and hence with (15) we have

$$Q\sigma_i\delta' = Q\theta_{f(i)}\tau_{f(i)}. \tag{16}$$

However, in order to prove the theorem we have to find a substitution δ such that $Q\sigma_i\delta = Q\theta_{f(i)}$. In order to define δ recall that τ_k is a Skolemizing substitution and hence can be written as

$$\tau_k = \{x_o \leftarrow a_o \mid o \in O\}$$

for some finite index set O and new constants a_o. In this case we can treat in the refutation D the a_o as new variables and define the substitutions

$$\tau_k^{-1} = \{a_o \leftarrow x_o \mid x_o \leftarrow a_o \in \tau_k\}.$$

Every τ_k introduces new Skolem constants. Hence the domains of the τ_k^{-1} are pairwise disjoint. But then with defining $\tau^{-1} := \tau_1^{-1} \ldots \tau_r^{-1}$ we get

$$\tau^{-1}|_{dom(\tau_k^{-1})} = \tau_k^{-1}. \tag{17}$$

Next define

$$\delta := \delta'\tau^{-1}.$$

This is the desired substitution since

$$Q\sigma_i\delta = Q\sigma_i\delta'\tau^{-1} \overset{(16)}{=} (Q\theta_{f(i)}\tau_{f(i)})\tau^{-1} \overset{(17)}{=} (Q\theta_{f(i)}\tau_{f(i)})\tau_{f(i)}^{-1} = Q\theta_{f(i)}.$$

Finally, with the fact that f is a suitable surjection the theorem is proved. □

2.3. The minimal answer problem

Certainly it is interesting to tune our procedure such that it computes minimal answers only. This means we want to be able to decide for a computed answer C of a clause set P, written $P \models C$, whether there is no other answer C' with $P \models C'$ that is more general than C. The decidability of this problem depends on the notion of "more general than" one wishes to implement. In the subsequent discussion, we will refer to the problem as the minimal answer problem.

We can interpret "more general than" as *subsumption*. A clause C' subsumes the clause C, written $C' \rhd C$, iff there is a substitution θ with $C'\theta \subseteq C$. Subsumption of clauses is an NP-complete problem [14]. Nevertheless it is decidable hence. However, the minimal answer problem is essentially undecidable, i.e., not even semi-decidable, because otherwise we could decide consistency, i.e., satisfiability of the clause set P as follows:

Consider the clause set $P \cup \{p\}$ where p is a nullary predicate symbol which does not occur in P. Then clearly P is satisfiable iff $P \cup \{p\}$ is. Since $P \cup \{p\} \models p$ trivially, p is a computed answer for $P \cup \{p\}$. Now we could ask whether there is another answer, i.e., set of literals, which subsumes p. This is only possible for the empty clause. So we would be able to decide whether $P \cup \{p\} \models \square$ does not hold, thus the satisfiability of $P \cup \{p\}$ and hence of P itself. But the satisfiability problem is known to be essentially undecidable for first-order clause sets.

We can try it with *logical implication* instead of subsumption, i.e., we want to find out whether there is no answer C' such that $C' \Rightarrow C$. But in this case, the minimal answer problem is again essentially undecidable, since implication is a notion more general than subsumption, i.e., $C' \rhd C$ implies $C' \Rightarrow C$. In contrast to the subsumption problem, already the implication problem for clauses is undecidable even for restricted cases; see [29].

A *condensation* of a clause C is a minimum cardinality subset $C' \subseteq C$ such that $C' \rhd C$. It holds that any condensation of the clause C is logically equivalent to C, and any two condensations of a clause are variants. This means we can restrict ourselves to computing only *condensed* answers C, i.e., C is its own condensation. Condensation is a local property because it does not take the clause set P into account. Since the clause condensation problem is decidable, the minimal answer problem becomes decidable too in this case. However, we refrained from implementing a condensation algorithm into our system because of its appalling complexity. Deciding whether a clause is condensed is co-NP-complete [12].

Nevertheless our calculus is answer-complete, i.e., for all correct answers C of a clause set P there is a subsuming computed answer C' such that $C' \rhd C$, although it may be the case that not all answers in condensed form can be computed. To see this, look again at Example 14. However, in Section 3 below we will describe a calculus variant which is more optimal with respect to the length of the disjuncts.

3. Definite answers and regularity

From theorem proving with ME we know that the regularity check is an important means for improving efficiency. Regularity for ordinary ME means that it is never necessary to construct a tableau where a literal occurs more than once along a branch. Expressed differently, it says that it is never necessary to repeat in a derivation a previously derived subgoal (viewing open leaves as subgoals). In this section we will present a variant of RME, the *ancestry restart* variant, which allows global regularity checks (see Definition 10). This variant is motivated by Loveland's UnH-Prolog [23].

As an interesting side effect it turns out that this variant offers considerable benefits with respect to logic programming. Occasionally one is interested in the question whether a given program with query admits a *definite* answer, i.e., an answer which is a single conjunction of atoms, but not a disjunction. Of course, in general, a non-definite program does not always admit a definite answer, but some programs do. It is the latter class of problems we are interested in now.

Example 18. Consider the program $P_{Def} = \{P(X,a) \vee P(b,Y) \leftarrow\}$ and the query $\leftarrow P(X,Y)$. Note that, among others, $P(b,a)$ and $P(X,a) \vee P(b,Y)$ are correct answers. Now, by Theorem 16 strict RME is answer-complete; hence we can find a refutation yielding a computed answer which entails $P(b,a)$ (Fig. 5, left side). As noted after the statement of Theorem 16, RME will not always compute *minimal* (with respect to length) answers. It is easily verified that in this example the shortest computed answer is $P(X,a) \vee P(b,Y)$, which can be factored to $P(b,a)$. This is due to repeated use of the query in the refutation.[4]

[4] However, in this example we could find a *non*-strict RME using the query only once. Hence, one might object that this example is vacuous. This, however, misses the point because there exists a more complicated example where this argumentation would not work.

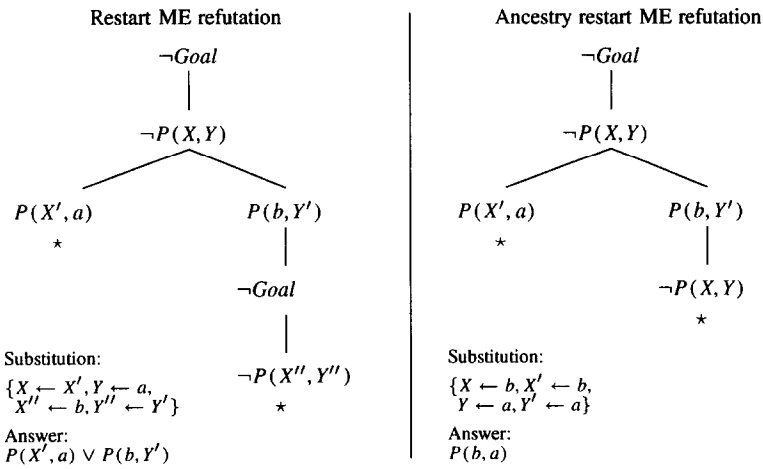

Fig. 5. An RME (left side) and an ARME (right side) refutation of $\{P(X,a) \vee P(b,Y) \leftarrow, \leftarrow Goal, Goal \leftarrow P(X,Y)\}$ depicted as trees.

The key idea to the direct computation of definite answers is to restrict the use of the query to one single application in the refutation, namely at its top. Then, by definition, definite answers are obtained. However, such a restriction is incomplete. But if RME is modified such that *every* negative literal along a branch, not only the topmost literal, may be used for the restart step then completeness is recovered. This follows from a more general result which states that we can restrict our calculus to *globally regular* refutations (i.e., no literal except the literal used for the restart occurs more than once along a branch). Let us now introduce all this more formally.

Definition 19 (*Ancestry restart model elimination* (*ARME*)). The calculus *ancestry RME* (*ARME*) is the same as *strict RME* (Definition 3), except that the inference rule *restart step* is replaced by the inference rule *ancestry restart step*:

$$\text{ancestry restart step:} \quad \frac{[p], \boldsymbol{P}}{[p] \circ \neg A, \boldsymbol{P}}$$

if *Leaf*($[p]$) is a positive literal and $\neg A \in [p]$.

The term "ancestry" in the definition is explained by the use of ancestor literals for restart steps. Note that any reduction from a *positive* leaf literal to a negative ancestor literal $\neg A$ can be simulated in ancestry RME by a restart step with $\neg A$, followed by a reduction step. Thus, non-strictness is "built into" ancestry RME. Note in addition, that the ancestry restart rule includes the restart rule since the first literal of the branch, which is always $\neg Goal$, can be used for the restart as well.

Fig. 5 (right side) shows an example ancestry RME refutation of P_{Def} with query $\leftarrow P(X,Y)$ for Example 18. Note that no new copy of the query clause is used, but instead the present instance is copied by the ancestry restart rule, and then the resulting branch is closed by a reduction step. This example also demonstrates that—unlike in

strict RME—it makes sense to apply a reduction step to a restart literal. This motivated us to change the definitions in [4] in now letting the restart step be an explicit inference rule.

Clearly, in terms of a proof procedure the ancestry restart rule induces a larger local search space than the restart rule. On the other hand, refutations may become much shorter. In order to see this think of a derivation containing a branch $[\neg B_1 \cdot \ldots \cdot \neg B_n \cdot A]$. It might be necessary in RME to repeat all the $\neg B_i$ in order to find a refutation. This derivation of $[\neg B_1 \cdot \ldots \cdot \neg B_n \cdot A \cdot \neg B_1 \cdot \ldots \cdot \neg B_n]$ can be abbreviated in ancestry RME to $[\neg B_1 \cdot \ldots \cdot \neg B_n \cdot A \cdot \neg B_n]$ by guessing the right $\neg B_n$ for the restart. Indeed, this is the rationale for our proof procedure to search the restart literals from the leaf towards the top. As a further benefit of this search order note that a definite answer will be enumerated *before* a non-definite answer, provided it allows for a shorter proof.

Ancestry RME has some similarities to Plaisted's *modified problem reduction format* (MPRF) [27]. Expressed in our terminology, MPRF corresponds roughly to the non-strict version of ancestry RME. As major differences we see that answer computation was not an issue in [27], and also that regularity refinements were not considered. These differences justify the need for our new proofs.

Now we proceed to an appropriate completeness result with respect to definite answers. As mentioned above, this result shall be a consequence of a more general result concerning a regularity restriction. Let us define this notion precisely:

Definition 20 (*Global regularity*). Let $[p]$ be a branch written as in Definition 10. The branch $[p]$ is called *globally regular* iff it is blockwise regular (conditions (i) and (ii) in Definition 10), and additionally

(iii) $B_i^l \neq B_j^m$ for $1 \leqslant l < m \leqslant n$, $1 \leqslant i \leqslant k_l$, $2 \leqslant j \leqslant k_m$ (*global negative regularity*)

holds. The notion of *global regularity* is extended towards branch sets and derivations as in Definition 10.

Hence we have three conditions, two from the definition of blockwise regularity and a third one from the above definition. Condition (i) states that all positive literals along a branch are pairwise different, and condition (ii) states that negative literals inside blocks are pairwise different, where by a block we mean a smallest subbranch delimited by positive literals or the ends of the branch. Condition (iii) means that a negative literal may be equal to one of its ancestors only if it follows a positive literal, i.e., if it is used as a restart literal. Thus we have a global regularity condition, except for restart literals. In all example refutations given so far, all branches are blockwise regular. However, the refutation in Fig. 1 (right side) is not globally regular, as can be seen by the two occurrences of $\neg Q$ in the rightmost branch. From this example we learn that RME is incompatible with the global regularity restriction. However we have:

Theorem 21 (Completeness of ARME). *Let c be a computation rule, f be a head selection function and S be an unsatisfiable clause set in Goal normal form. Then there exists a globally regular, ancestry RME refutation of S with computation rule c and head selection function f.*

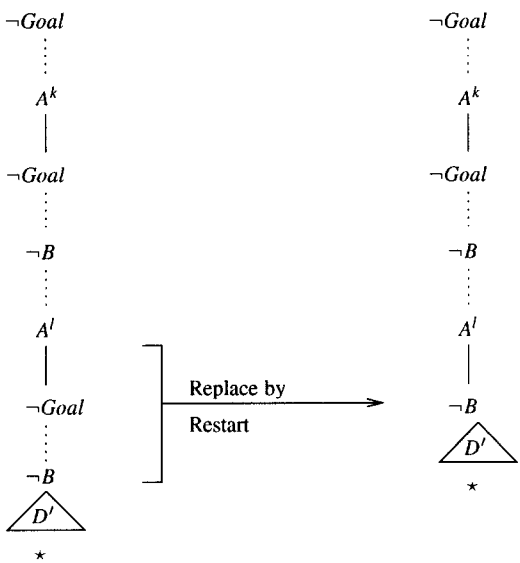

Fig. 6. Transformation step removing a violation of a global negative regularity constraint.

Proof. The proof builds on the above answer-completeness Theorem 16, which in turn relies on the ground completeness Theorem 12. We can thus rely on this result, and show how to transform a given blockwise regular refutation in RME on the ground level to a globally regular refutation in ancestry RME (also on the ground level).

This suffices to prove the theorem on the first-order level for the following reason: suppose some unsatisfiable clause set in goal normal form is given. From the cited results we know that there exists a strict blockwise regular RME refutation, say D, which is lifted from a strict blockwise regular RME refutation D^g on the ground level. By the transformation proposed below then there exists a globally regular ARME refutation D^g_{Anc}. It is straightforward to adapt the lifting theorem (Theorem 15) to take into account the ancestry restart step. The adaption to global regularity instead of blockwise regularity is straightforward as well. Hence D_{Anc} is also globally regular.

It remains to prove the desired transformation. For this let D^g be the given strict, blockwise regular RME refutation from above. Since by definition an ancestry restart step includes the possibility of a restart step, D^g is also a blockwise regular ARME refutation. Let n be the number of violations of the global negative regularity constraint (condition (iii) in Definition 20). Either $n = 0$ and we are done, or else D^g can be transformed to contain strictly less than n such violations, without sacrificing the blockwise regularity constraints. Repeated application will eventually result in the desired refutation.

The transformation step can most easily be expressed using the tree view of ME. Fig. 6 (left) shows the general situation: assume D^g contains a branch

$$[p] = [\neg Goal \cdot \ldots \cdot A^k \cdot \neg Goal \cdot \ldots \cdot \neg B \cdot \ldots \cdot A^l \cdot \neg Goal \cdot \ldots \cdot \neg B],$$

where A^l is the bottommost (rightmost) positive literal dominating $\neg B$. Further, as indicated by the two occurrences of $\neg B$ in $[p]$, the global negative regularity constraint is violated for $[p]$. Let D' be the tableau corresponding to the refutation of the branch $[p]$.

Now delete the subtree between A^l and $\neg B$ below (bounds excluded from deletion). We arrive at a new tableau with branch

$$[p'] = [\neg Goal \cdot \ldots \cdot A^k \cdot \neg Goal \cdot \ldots \cdot \neg B \cdot \ldots \cdot A^l \cdot \neg B].$$

Next append D' to $[p']$. Since *only negative* literals have been deleted, and we assume that D^g is a *strict* RME refutation, in D' no reduction steps from positive literals to negative literals have occurred. Thus D' is also a closed tableau below $[p']$ (right side in Fig. 6).

It is clear from the definition that this new tableau can be constructed by means of an ancestry restart step, using $\neg B$ as restart literal. Furthermore, (at least) one violation of the global negative regularity constraint has been removed, and since we only have deleted some literals from p the blockwise regularity constraints will not be violated. Thus we have found a suitable transformation step. □

We can use this result to obtain the desired completeness result for definite answers.

Theorem 22 (Answer-completeness of ARME). *Ancestry RME is answer-complete in the sense of Theorem* 16. *In particular, if $Q\theta$ is a correct definite answer for a program P, then there exists an ARME refutation of P with computed answer $Q\sigma$ such that $Q\sigma\delta = Q\theta$, for some substitution δ. Furthermore, the input clause $Goal \leftarrow Q$ is used exactly once, namely as the goal clause of the refutation.*

This last theorem enables us to enumerate definite answers *only*, by simply restricting the use of $Goal \leftarrow Q$ to one extension step at the beginning. So we have the desirable properties of loop checking by regularity and the computation of definite answers.

Note 1 (*Restriction to programs*). Notice that in Theorem 22 we have one important change now with respect to the results so far: for the special case of singleton definite answers we require our clause sets to be *programs*, i.e., no negative clauses are allowed (except the query, of course). This restriction can be explained using the following example.

Example 23. Consider the clause set P and the query $\leftarrow Q$:

$$\begin{array}{lll} P: & P(X,Y), Q(X,Y), Q(Y,X) \leftarrow & \\ & & \leftarrow P(X,Y) \\ \leftarrow Q: & & \leftarrow Q(X,Y). \end{array}$$

It is clear that $Q(X,X)$ is a correct (definite) answer. However, with the "wrong" head selection function which selects $P(X,Y)$ in the first clause it is easy to see that neither an RME nor an ARME refutation of P with goal clause Q exists. The problem

is the incompatibility of the concepts "head selection function" and what could be called "independence of the goal clause", i.e., that the goal clause can be chosen freely among all negative input clauses.

Of course, there exist refutations with the goal clause $Goal \leftarrow P(X,Y)$, stemming from the second clause of P. However, the proof of Theorem 22 for the case of definite answers relies on the fact that the *query* is used as the goal clause of the ARME refutation (i.e., on top of the tree). However, in presence of negative clauses in the program this can in general not be achieved, and hence Theorem 22 does not hold in such a setting. Thus, restricting to *programs* solves the problem, because then there is only one negative clause—the query—and thus we are forced to use it as the goal clause.

On the other side, if the "selection function" is given up, i.e., any positive literal in a clause may be accessed as an entry point, then the "independence of the goal clause" holds. Consequently, Theorem 22 can be recovered in this setting. However, a detailed treatment is beyong the scope of this paper and will be presented elsewhere.

Proof. The answer-completeness follows directly from the fact that the ancestry variant still permits restart steps with the goal, i.e., it allows for additional derivations. The proof of the last part is given by a careful analysis of the proof of Theorem 16. Recall from that proof that $Q\theta$ is a correct answer because P implies that $S' = P \cup \{\neg Q\theta\tau\}$ is unsatisfiable, where τ is some substitution introducing new Skolem constants for the variables of $Q\theta$. Then we considered a set S'''' of ground instances of the goal normal form of S'. It is important to recognize that the goal normal form S'''' contains exactly one instance of the query, namely $Goal \leftarrow Q\theta\tau$. Since we assume that P is a *program*, and hence consistent, it follows from the proof of Theorem 12 that there exists a strict RME refutation of S'''' where the first extension step is done with the clause $Goal \leftarrow Q\theta\tau$. In other words, every branch in the refutation is of the form

$$[\neg Goal \cdot \neg Q_i\theta\tau \cdot \ldots]$$

where Q_i is some literal in Q. Recall that Q is an abbreviation for $Q_1 \wedge \cdots \wedge Q_n$. Now suppose that $Goal \leftarrow Q\theta\tau$ is used once again in the refutation. For syntactical reasons this can happen only if $Goal \leftarrow Q\theta\tau$ is extended to a branch resulting from a restart step. This branch takes the form

$$[\neg Goal \cdot \neg Q_i\theta\tau \cdot \ldots \cdot A \cdot \neg Goal]$$

where A is some positive literal. Extension with $Goal \leftarrow Q\theta\tau$ results (among possible others) in a branch

$$[\neg Goal \cdot \neg Q_i\theta\tau \cdot \ldots \cdot A \cdot \neg Goal \cdot \neg Q_i\theta\tau].$$

But now note that this extension step leads to a violation of the global negative regularity restriction and thus will be eliminated by the transformation given in the proof of Theorem 21. To be more precise, the branch

$$[\neg Goal \cdot \neg Q_i\theta\tau \cdot \ldots \cdot A \cdot \neg Q_i\theta\tau]$$

will come up instead. Since every extension with $Goal \leftarrow Q\theta\tau$ will be eliminated in this way, the query clause $Goal \leftarrow Q\theta\tau$ is used precisely once in this transformed refutation, namely for the first extension step. The lifting argument for this refutation then is the same as in Theorem 16. Thus we arrive at a refutation with computed answer $Q\sigma$ which is more general than $Q\theta$. This completes the proof. □

4. Implementation

All variants and refinements of ME discussed so far are implemented in the PROTEIN system [5]. PROTEIN is a first-order theorem proving system based on the PROLOG technology theorem proving (PTTP) technique [32], implemented in ECLiPSe-PROLOG [10]. The general idea of the PTTP-implementation technique is to view PROLOG as an almost complete theorem prover that has to be extended by only few ingredients in order to handle the non-Horn case. More precisely, the given clause set is compiled into a PROLOG program whose execution is a search for a refutation of the given input clause set. We will now state some of PROTEIN's features in more detail.

4.1. The PTTP technique

Since in general the input clause set is non-Horn, the resulting compiled PROLOG program will have some extra facilities corresponding to the deviation of the calculus under consideration from PROLOG's SLD resolution. Two main extensions are the introduction of negative predicates, in order to handle contrapositives whenever this is necessary, and the introduction of ancestry lists, which are used to store the information contained in branches. The latter allows the compiler to insert the code for reduction and ancestry restart steps.

By the PTTP technique we get a rapid and flexible, but nevertheless efficient system which hosts both full predicate (clause) logic and answer computing. These properties—flexibility and rapid prototyping—are the main advantages of the compilation to PROLOG over a dedicated extended WAM approach. So, the WAM technology and other benefits of optimising PROLOG compilers are accessible to theorem proving and disjunctive logic programming.

However, the use of a cut or other non-logical extensions of PROLOG should be avoided. Nevertheless it is possible to integrate plain PROLOG code. This is especially useful if parts of the problem description do not need the full power of first-order predicate logic, or if one wants to exploit the procedural, non-logical capabilities of PROLOG.

4.2. Computing answers with PROTEIN

Since ME is a goal-oriented, linear, and answer-complete calculus, it is really well suited as an interpreter for disjunctive logic programming. PROTEIN facilitates computing disjunctive and definite answers with respect to positive disjunctive logic programs.

In the newest PROTEIN release there is a flag which allows us to look for definite answers only.

In our implementation, the definiteness of the answer can be guaranteed by the fact that all occurrences of answer literals during the proof search are unified with each other. This is the approach taken for resolution in [28]. It is applicable in every variant of ME, not only in the ancestry variant. For this, coroutining techniques are used.

But it is also possible to force our system to use the query only once as the goal clause according to Theorem 22. When using this setting for *clause sets* (as opposed to programs) such as those of the next section, the head selection function feature has to be disabled in order to preserve completeness (cf. Note 1).

4.3. Additional features

Different search strategies are built into PROTEIN, e.g. iterative deepening, depth-first or more general weighted search. The restart, strict, and ancestry variants (possibly with selection function), loop checking by regularity and also factorization (which is introduced in [20]) are parts of the system.

Another distinguished feature of PROTEIN is its theory interface (PROTEIN = PROver with a Theory Extension INterface). PROTEIN includes *theory reasoning* [1,2,31] in a very general way. Theory reasoning allows a calculus to relieve from explicit reasoning in some domain (e.g. equality, partial orders, taxonomic reasoning) by taking apart the domain knowledge and treating it by special inference rules. In an implementation, this results in a universal "foreground" reasoner that calls a specialized "background" reasoner for theory reasoning.

Another instantiation of this approach allows us to plug in constraint reasoning components, thus making available the ECLiPSe-PROLOG constraint solving mechanism in a sound and complete manner for theorem proving [7].

5. Comparative theorem prover study

In the sequel, we want to relate our experiences in computing answers by using theorem provers. First of all, we had to overcome some technical problems because theorem provers usually do not supply answers apart from "yes" or (possibly) "no". We will illustrate our experiences with a puzzle example which allows for both indefinite and definite answers. We will discuss this example in detail, but we tested many other examples. All considered examples are contained in the TPTP problem library [33] and were used without changes in our experiments.

5.1. Knights and knaves

Example 24. Our main example follows problem #36 in [30]. A related example is studied in [26]. The manner of formalizing problems there is similar to ours. The natural language description of our problem is given in Fig. 7.

Natural language description:

(i) *On an island, there live exactly two types of people: knights and knaves.*
(ii) *Knights always tell the truth and knaves always lie.*
(iii) *I landed on the island, met two inhabitants, asked one of them: "Is one of you a knight?" and he answered me.*
(iv) *What can be said about the types of the asked and the other person depending on the answer I get?*
(v) *We assume, that either a proposition or its negation is true.*
(vi) *If the disjunction of two propositions is true then at least one of them must be true.*

Formulae set with two cases:

(i) $true(isa(Q, knight)) \dot{\vee} true(isa(Q, knave))$.
(ii) $says(P, S) \rightarrow (true(S) \leftrightarrow true(isa(P, knight)))$.
(iii) (a) $says(asked, \bullet)$ ("yes"),
(b) $says(asked, not(\bullet))$ ("no")
where $\bullet = or(isa(asked, knight), isa(other, knight))$.
(iv) $\neg(true(isa(asked, X)) \wedge true(isa(other, Y)))$.
(v) $true(not(C)) \dot{\vee} true(C)$.
(vi) $true(or(A, B)) \leftrightarrow (true(A) \vee true(B)))$.

Possible answers:

- *knave/knave* $\vee$ *knave/knight* $\vee$ *knight/knave* $\vee$ *knight/knight* (trivial).
- *knave/knave* $\vee$ *knight/knave* $\vee$ *knight/knight* (indefinite, case (a) only).
- *knave/knight* (definite, case (b) only).

Fig. 7. Smullyan's problem #36.

The reader may think about the problem for a while, before he looks at the solution which is given next. The last two pieces of information (v) and (vi) explicitly state some knowledge about inferencing. We need them in order to be able to cope with the information in (ii) because our description language is only first-order. It is interesting to note that a human being usually knows both these facts implicitly. We have to make a case distinction:

(a) Suppose, the asked person answers with "yes". Then he may be a knight, because then of course it is true that one of them is a knight. In this case, the other person can be a knight or a knave. We cannot even exclude that the asked person is a knave. Then it must be true that none of them is a knight, hence the other person is also a knave in this case.
(b) Let us now assume, the asked person answers with "no". Then this person must be a knave, since a knight cannot answer with "no" honestly. It follows, that the other person is a knight, because one of them must be a knight. So, in this case we get a definite answer.

In our formalization of the problem (also given in Fig. 7), the formulae in (i) and (ii) express the corresponding pieces of information from the natural language description.

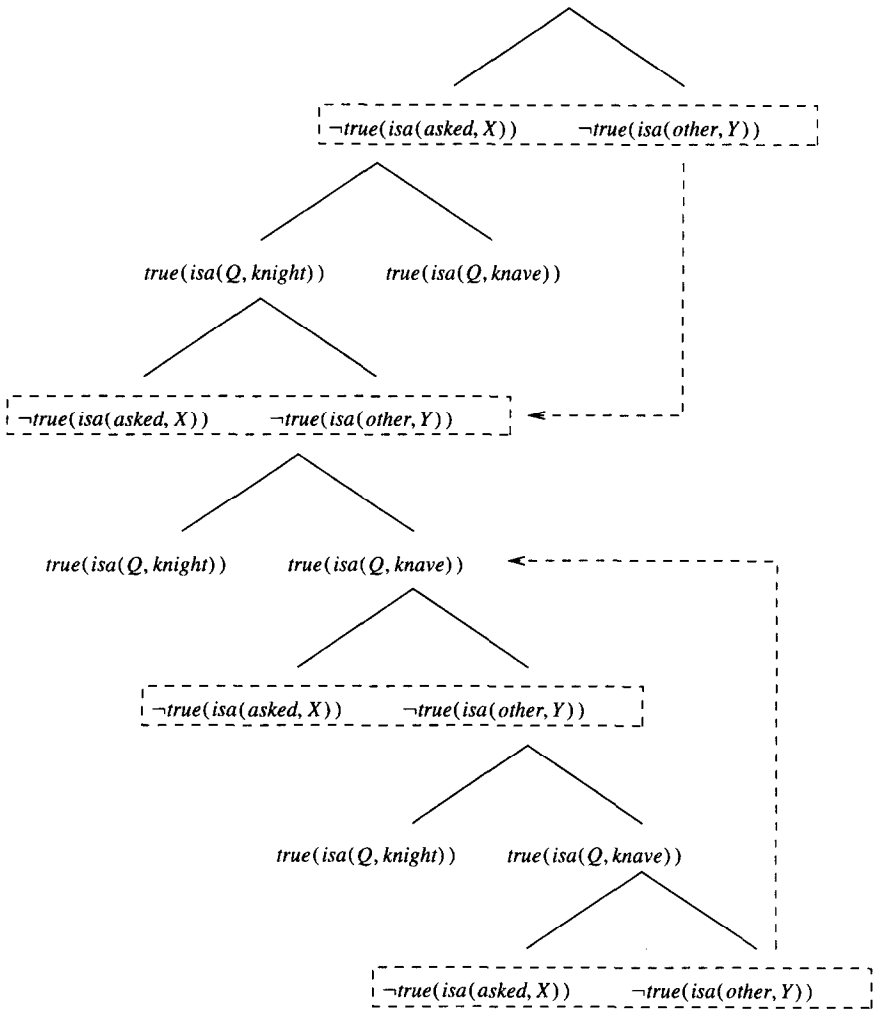

Fig. 8. Derivation of the trivial answer.

Depending on the case considered, we choose exactly one of the formulae (iii)(a) or (iii)(b). This means, we actually split the problem into two formulae sets. We view the fact that a person denies a question as that he says that the thing in question is not true using the binary predicate *says* (instead of a ternary predicate). Formula (iv) can be considered as the query; it is negated because we do refutation-based reasoning. We have to express the pieces of information (v) and (vi) explicitly by introducing the unary predicate *true*. The symbol $\dot{\vee}$ denotes *exclusive or*. The transformations of the formulae below into clausal form are straightforward and therefore omitted here. (But look at Fig. 11.) They consist of 11 clauses.

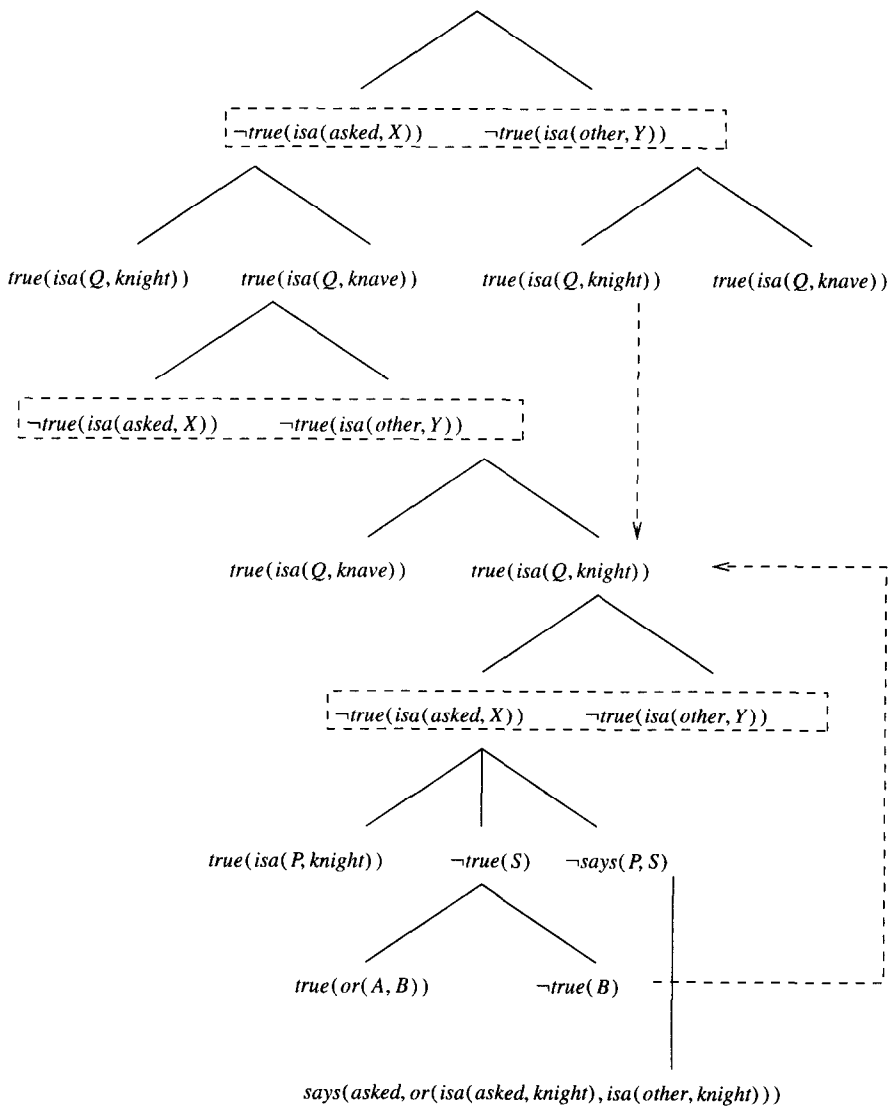

Fig. 9. Derivation of the indefinite answer (case (a) only).

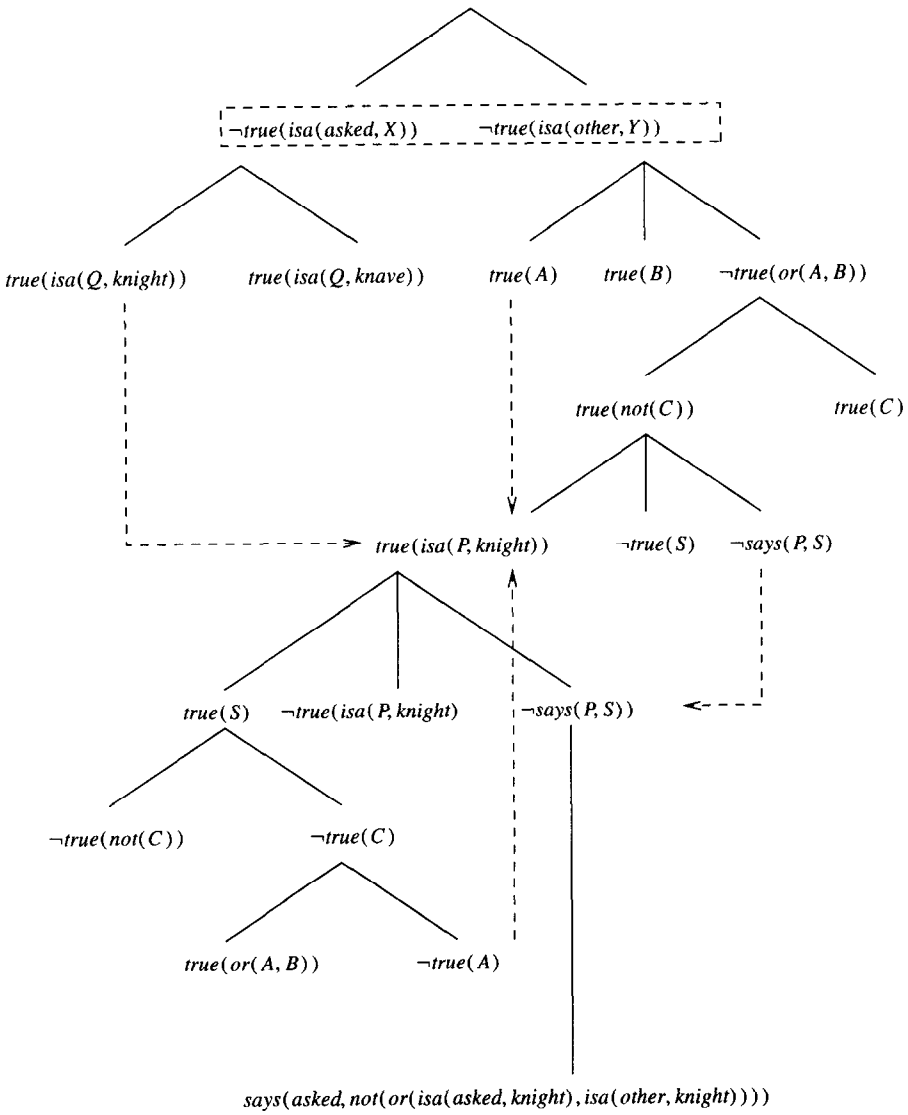

Fig. 10. Derivation of the definite answer (case (b) only).

We can prove the query in many different ways. As a consequence we get many trivial and hence useless answers. The (most) trivial one—a four part disjunction—can be obtained in both cases. We only need formula (i) and the query in order to infer it. But it only says that each of both persons are either knights or knaves. In case (a) (if the asked person says "yes") we can get an indefinite answer consisting of only three disjuncts. In the other case (b) (answer "no") there exists a definite answer. Fig. 7 contains a list of these possible answers where X/Y is an abbreviation of $true(isa(asked, X)) \wedge true(isa(other, Y))$.

Figs. 8, 9 and 10 show tableaux for the derivations of the trivial, indefinite and definite answer respectively. All occurrences of the query are emphasized with dashed boxes. Substitutions are not annotated in order to keep the presentation clearer. Dashed upward links denote reduction steps. Dashed downward links abbreviate the presentation. They indicate proof parts that can be used repeatedly which can also be explained by factorization [20]. Before turning to our experiments we want to mention some interesting facts. They indicate the difficulties of finding the precise answers by means of theorem proving systems.

- Answer-completeness requires that we are able to compute the indefinite and definite answers in the respective cases for Example 24. In order to be able to derive these answers we need a clause set which is not minimally unsatisfiable; note that the clauses of (i) and (iv) together are (minimally) unsatisfiable yielding the trivial answer (see Fig. 8).
- Nine steps are needed to derive the indefinite or the definite answer respectively, while only seven steps are needed to derive the trivial answer (in both cases). So we expect the more precise answers to be found later during the proof search. This turns out to be true for general resolution-based approaches, as the run time results indicate. But our restart variants, especially ancestry RME, behave superior since they prefer shorter answers.

5.2. *Experimental results*

We tried to get the answers from Fig. 7 automatically by using the theorem proving systems listed below. We used the clause ordering given by the problem description. Nevertheless our experiments show that the (run time) results depend on the ordering. All experiments have been performed on a Sparc station 10/30 with 32 MB RAM and about 50 Mips processor rate. We investigated more puzzle examples from [30] as formulated in the TPTP problem library [33] that allow for definite answers. In addition, we tried some planning examples and so-called blind hand problems, because both classes represent problems where answers besides "yes" or "no" are really desirable. A short description list of the theorem provers taken into consideration follows.

- *OTTER* [25]: resolution-style theorem proving program coded in C for first-order logic (with equality).
- *SETHEO* [16]: top-down prover for first-order predicate logic based on the calculus of the so-called connection tableaux which generalizes ME; it uses a WAM compiler similar to PROLOG, the resulting code is interpreted by a C program.

Table 1
Run time results I

TPTP	NAME	ANSWER	SETTING	OTTER		SETHEO	PROTEIN			SATCHMO
				binary	hyper	default	ME	RME	ARME	level
PUZ023-1	knight/knave #27	definite	trivial	0.12	0.13	0.32	0.35	∞	∞	∞
			general	∞	†	/	∞	∞	∞	0.30
			special	/	/	11.40	∞	∞	∞	0.30
PUZ025-1	knight/knave #35	definite	trivial	323.79	0.43	0.25	0.87	10.68	12.35	∞
			general	320.47	†	/	∞	128.23	157.05	768.60
			special	/	/	9.32	∞	119.57	47.40	766.07
PUZ026-1	knight/knave #39	definite	trivial	72.52	2.10	1.53	∞	∞	∞	*
			general	∞	2.05	/	∞	∞	∞	1.68
			special	/	/	11.73	∞	∞	∞	1.70
PUZ027-1	knight/knave #42	definite	trivial	0.53	0.70	1.67	112.67	∞	∞	∞
			general	∞	†	/	∞	∞	∞	0.57
			special	/	/	6.73	∞	∞	∞	0.55
PUZ035-1	knight/knave #36/1	indefinite	trivial	1.05	0.18	0.04	0.30	0.48	1.27	/
			general	†	†	†	0.13	0.35	0.45	0.03
			special	/	/	!	!	∞	∞	0.07
PUZ035-2	knight/knave #36/1	indefinite	trivial	0.92	0.21	0.08	0.77	0.55	1.10	/
			general	*	†	†	0.17	0.50	0.58	0.15
			special	/	/	!	!	∞	∞	0.20
PUZ035-5	knight/knave #36/3(a)	indefinite	trivial	0.62	0.06	/	∞	∞	46.27	0.10
			general	†	∞	0.15	1.97	414.77	308.90	∞
			special	/	/	∞	∞	∞	∞	∞
PUZ035-6	knight/knave #36/3(b)	definite	trivial	16.67	0.99	0.18	4.27	12.48	15.67	0.10
			general	†	†	/	∞	11.33	13.95	*
			special	/	/	0.72	∞	12.28	10.92	*
PUZ035-7	knight/knave #36	indefinite	trivial	∞	∞	/	∞	192.92	189.17	/
			general	∞	∞	1.83	∞	∞	∞	∞
			special	/	/	∞	∞	∞	∞	∞

- *SATCHMO* [24]: theorem prover consisting of just a few lines of PROLOG code; it is based on a model generation paradigm and refutationally complete if used in a level-saturation manner.
- *PROTEIN* [5]: as described in this paper; see especially Section 4.

Tables 1 and 2 summarize our experiments. The tables give the TPTP description and names of the problem; our Example 24 has the identifiers PUZ035-5 and PUZ035-6 for the cases (a) and (b), respectively. The next column shows what kind of answer we expect: a definite or an indefinite answer; the word "Horn" indicates pure Horn problems which clearly allow definite answers only. Another column titled "setting" contains three different entries: "general" means that a system setting is used which also derives indefinite answers, whereas "special" states settings which look for definite answers only; the rows preceded with "trivial" show the run times until the first trivial (indefinite) answer is found. Then the run times themselves follow, all of them are

Table 2
Run time results II

TPTP	NAME	ANSWER	SETTING	OTTER		SETHEO	PROTEIN			SATCHMO
				binary	hyper	default	ME	RME	ARME	level
MSC001-1	blind hand #1	definite	general	∞	7.93	32.77	1.85	1.31	1.68	*
			special	/	/	22.43	1.93	1.45	1.67	*
MSC002-1	blind hand #2	definite	general	720.84	7.74	0.18	0.13	0.65	0.60	*
			special	/	/	0.12	0.12	0.70	0.62	*
MSC002-2	blind hand #2	definite	general	660.89	7.54	0.12	0.12	0.47	0.48	*
			special	/	/	0.12	0.07	0.47	0.47	*
PLA001-1	buying bread	Horn	general	1.14	86.76	0.95	1.25	1.33	1.25	∞
			special	/	/	1.00	1.27	1.33	1.28	∞
PLA002-1	all weather	definite	trivial	0.17	0.95	0.08	0.15	0.65	0.70	*
			general	†	∞	∞	∞	∞	∞	*
			special	/	/	∞	∞	∞	∞	*
PLA002-2	all weather	definite	trivial	∞	∞	0.63	∞	8.75	12.03	*
			general	∞	∞	∞	∞	∞	∞	∞
			special	/	/	∞	∞	∞	∞	∞
PLA003-1	monkey-banana	Horn	general	0.16	0.15	0.02	0.10	0.08	0.10	0.07
			special	/	/	0.12	0.08	0.10	0.10	71.77
PLA004-1	block's world #1	Horn	general	∞	∞	0.87	0.57	0.60	0.55	*
			special	/	/	1.10	0.55	0.62	0.55	*
PLA004-2	block's world #1	Horn	general	∞	∞	0.10	0.70	0.80	0.73	*
			special	/	/	0.38	0.73	0.83	0.73	*
PLA005-1	block's world #2	Horn	general	∞	∞	0.18	0.15	0.15	0.15	*
			special	/	/	0.18	0.17	0.15	0.15	*
PLA005-2	block's world #2	Horn	general	∞	∞	0.15	0.07	0.07	0.07	*
			special	/	/	0.15	0.07	0.10	0.05	*
percentage of solved problems in total				25%	30%	75%	60%	70%	70%	35%

given in seconds. "∞" denotes the fact that we get no answer after 900s; a slash "/" means that the respective setting is not applicable; if the memory is exhausted then we write "*". The other symbols will be explained in the sequel.

5.3. *Resolution- and ME-based systems*

We used OTTER in settings that are complete with respect to finding refutations, namely binary and hyper-resolution (plus factorization). All clauses are put into the "set of support" [8]. But OTTER has some problems with computing answers because it enumerates resolvents but not all (refutational) proofs. Especially during the subsumption test, it did not take the answer literals into account which are provided for computing answers. That is the reason why OTTER with (forward and backward) subsumption is *not* answer-complete.

An example which illustrates this, is case (b) (PUZ035-6) where the search stops after finding many times only the trivial answer with binary resolution. Such cases are marked with †. We got similar problems with case (a) (PUZ035-5) and binary resolution. However, we find a proof by using hyper-resolution plus factorization immediately. We also tried the autonomous mode of OTTER. But it does not solve more problems than the other modes, and in autonomous mode, completeness is not guaranteed.

In theory, there is a solution to the problem with subsumption. It can be shown that we only have to take into account the answer literals during the subsumption steps. This can easily be shown by extending the argument in the completeness proof of the resolution principle in [8] where the answer literals have to be treated explicitly. Unfortunately, it is not (yet) possible to test OTTER in this setting and find out whether this improves the behaviour, because it is not built in.

We generate answers with SETHEO by using global variables. The answers are kept in a list. By this and other technical tricks, we find the indefinite answer for case (a) (PUZ035-5) within 0.15s and the definite answer within 0.18s. That is quite good and may be explained by the subgoal reordering heuristics built into SETHEO, which are not (yet) incorporated into our system. However, SETHEO also has constraints which are used in the default setting. Unfortunately, the anti-lemma constraints [15] destroy answer-completeness in SETHEO. Examples for this are marked with a †, e.g. PUZ035-1 and PUZ035-2 (which will be discussed in Section 5.4).

PROTEIN *is* answer-complete; that has been stated in this paper. It finds the indefinite and definite answers for the respective case. We tried both, plain and restart ME. In the case of the restart variant we also tried its refinements: with or without ancestry restart or selection function (no contrapositives). We tried to compute the desired answers with settings where all solutions are computed in case (a) (indefinite answer). For the case (b) (definite answer) we used the setting where only definite answers are searched for.

As one can see, using restart helps for case (b) (but it is not advantageous for case (a)), since plain ME does not find the desired answer, although it does so for trivial answers. In most cases, trivial answers are found earlier than the other more specific ones. In some cases, we are able to find out that there is no definite answer which is indicated with "!" (again for PUZ035-1 and PUZ035-2, also with SETHEO in the "special" setting). The reader may look at the rest of the table on his own.

5.4. *Model elimination versus model generation*

It seems that a model generation approach is very adequate for puzzles because they often allow for finite models. In this special case we can derive the answers from the models by the following non-deterministic procedure: generate all models of the given clause set, e.g. with SATCHMO, extract from each of the (finitely many) models one instance of the query, and combine them into one disjunctive answer. This is the "special" setting for SATCHMO. The proof that justifies this procedure is trivial for the finite case and hence left out here. We use SATCHMO only with level-saturation, because the basic procedure is not refutationally complete and does not solve more puzzles.

It is important to notice that SATCHMO needs range-restricted input, i.e., any variable occurring in the head of a clause must also occur in the body. Any clause set can be translated into range-restricted form that is satisfiable iff the original one is so. For this purpose, an auxiliary predicate *dom* is introduced, and the following transformations are performed, according to [24]:

- For each variable x which occurs not range restricted in some rule, the subgoal $dom(x)$ is inserted in the body of this rule.
- For every n-ary ($n \geqslant 0$) function symbol f, we add the clause $dom(f(x_1,\ldots,x_n)) \leftarrow dom(x_1) \wedge \cdots \wedge dom(x_n)$. If the clause set does not contain any constant, a single clause $dom(a)$ is added where a is a new constant symbol.

However, if this transformation is performed literally on Smullyan's puzzles #27, #35, #39 and #42, then we do *not* find any of the answers at all. The reason is that the above-stated translation is very naive. It does not take into account that the problem domains are somehow naturally structured. Let us consider for example the first clause from Example 24:

$$true(isa(Q,knight)) \vee true(isa(Q,knave)).$$

By inspection of the whole clause set, we realize that Q can only be instantiated with *asked* or *other*. But the naive transformation yields

$$true(isa(Q,knight)) \vee true(isa(Q,knave)) \leftarrow dom(Q)$$

where the *dom* predicate has to be defined as follows:

$$\begin{array}{llll} dom(asked). & dom(other). & & \\ dom(knight). & dom(knave). & & \\ dom(isa(P,Q)) & \leftarrow & dom(P), dom(Q). & \\ dom(not(C)) & \leftarrow & dom(C). & \\ dom(or(A,B)) & \leftarrow & dom(A) \wedge dom(B). & \end{array}$$

By this, Q will be instantiated by compound terms, such as e.g. *isa*(*asked*, *not*(*knave*)), which is obviously unnecessary and in addition semantically nonsense.

So we propose an extended procedure for automatically transforming clause sets into range-restricted form. The main idea is to replace the single *dom* predicate by a set of specialized *dom* predicates. Of course, the following procedure cannot decide which terms will occur on which argument positions, but it gives a safe approximation:

Definition 25 (*Optimized range-restriction*). Let Σ_F and Σ_P be the sets of function and predicate symbols respectively, occurring in the given clause set S. Then we define $\Sigma = \Sigma_F \cup \{s.k \mid s \in \Sigma_F \cup \Sigma_P, 1 \leqslant k \leqslant n\}$ where n denotes the arity of the symbol s. Let now $\equiv$ denote the smallest equivalence relation on S which satisfies the following conditions:

- If $s(t_1,\ldots,t_n)$ is an atom or term occurring in S, and t_k ($1 \leqslant i \leqslant n$) is a term with main functor f, then $f \equiv s.k$.

(i) $true(isa(Q, knight)) \vee true(isa(Q, knave)) \leftarrow dom\_person(Q)$.
$\leftarrow true(isa(Q, knight)) \wedge true(isa(Q, knave))$.
(ii) $true(S) \leftarrow true(isa(P, knight)) \wedge says(P, S)$.
$true(isa(P, knight)) \leftarrow true(S) \wedge says(P, S)$.
(iii) (a) $says(asked, \bullet)$ ("yes").
(b) $says(asked, not(\bullet))$ ("no").
where $\bullet = or(isa(asked, knight), isa(other, knight))$.
(iv) $\leftarrow true(isa(asked, X)) \wedge true(isa(other, Y))$.
(v) $true(not(C)) \vee true(C) \leftarrow dom\_formula(C)$.
$\leftarrow true(not(C)) \wedge true(C)$.
(vi) $true(or(A, B)) \leftarrow true(A) \wedge dom\_formula(B)$.
$true(or(A, B)) \leftarrow true(B) \wedge dom\_formula(A)$.
$true(A) \vee true(B) \leftarrow true(or(A, B))$.
$dom\_person(asked)$. $dom\_person(other)$.
$dom\_type(knight)$. $dom\_type(knave)$.
$dom\_formula(isa(P, Q)) \leftarrow dom\_person(P), dom\_type(Q)$.
$dom\_formula(not(C)) \leftarrow dom\_formula(C)$.
$dom\_formula(or(A, B)) \leftarrow dom\_formula(A) \wedge dom\_formula(B)$.

Fig. 11. Range-restricted form for Smullyan's problem #36.

- If a variable x occurs in one clause of S as the ith argument of the symbol s and as the jth argument of the symbol t, then $s.i \equiv t.j$.

Any equivalence class with constant symbol in it has to be extended by an artificial new constant. For each equivalence class $[s]$ of $\equiv$, we now introduce a predicate $dom[s]$. For each function symbol $f \in \Sigma_F$ we add the predicate definition $dom[f](f(x_1, \ldots, x_n)) \leftarrow dom[f.1](x_1), \ldots, dom[f.n](x_n)$ to S. If the variable x occurs only in the head of a rule, and x is the ith argument of the symbol s, then add $dom[s.i](x)$ in the body.

By this procedure, we can identify three equivalence classes of $\equiv$ for Example 24, namely

(i) $person = \{isa.1, asked, other, says.1\}$,
(ii) $type = \{isa.2, knight, knave\}$, and
(iii) $formula = \{isa, not, not.1, or, or.1, or.2, says.2, true.1\}$.

This means the optimized transformation into range-restricted form looks as in Fig. 11. In the ordinary transformation of this example, there is only one *dom* predicate. But as the subsequent discussion shows, the optimized transformation does not help for this example to find the desired answers.

With SATCHMO [24] and the optimization just presented, we get all answers for the puzzle examples quite fast by using it with model generation since all considered puzzles allow for finite models—except #36. This is clear because in our formulations (PUZ035-5 and PUZ036-6) only infinite models exist. As soon as $true(S)$ is valid for some expression S, it must hold $true(not(not(S)))$ too; that means, we can derive facts

with any number of double negations. This increases the search space to such a degree that the refutationally complete level-saturation version of SATCHMO is not able to find the desired answers.

Therefore, we investigated several other formalizations of Smullyan's problem #36 (due to François Bry) which have finite models and are a bit longer in formulation than ours. Both versions permit the coding of the problem into one single clause set. Concerning version 1 (PUZ035-1 and PUZ035-2), SATCHMO finds the desired (disjunctive) answer within 0.03s. But PROTEIN also finds the answer within this time if we start in proof tree depth 6. Version 2 (PUZ035-3 and PUZ035-4, not listed in our tables) is a bit tricky since it uses the SATCHMO rule $(false \leftarrow S) \leftarrow true(not(S))$ which mixes object- and meta-levels. On the one hand, this is an advantage for SATCHMO; but on the other hand, it is not quite clear how then completeness can be assured. Ordinary first-order theorem provers need three additional lemmata in order to be able to find out the solution.

We developed a related variant of ME, called hyper-tableau [6]. It combines ideas from hyper-resolution and from analytic tableaux. It does not require range-restricted input sets. Thus, it mostly avoids enumerating terms which is the effect of the *dom* predicate.

6. Conclusion

To conclude, it seems to be very promising to use ME as a base calculus for computing answers in disjunctive logic programming. In this paper, we introduce (among others) the ancestry restart variant which is quite well suited for this purpose. We also give some practical evidence. Nevertheless, further investigation is necessary in order to find yet more efficient calculi. All our experiments corroborate the following facts:

- Resolution has difficulties in solving puzzles because of some problems with subsumption. Because of this, OTTER is not answer-complete. Only 30% of the considered problems could be solved by OTTER.
- Model elimination (which of course is related to resolution because it could be seen as linear resolution plus reduction steps) is better suited although it could not solve all puzzles that we tested. Nevertheless, SETHEO solves more problems than PROTEIN. There may be several reasons for this: SETHEO is efficiently implemented in C and has some more refinements, such as folding-up and folding-down [15], integrated in the system, whereas PROTEIN is an experimental system designed for rapid prototyping. In addition, SETHEO turned out not to be answer-complete as the examples PUZ035-1 and PUZ035-2 reveal.
- Model generation seems to be promising only for examples with finite domains (and hence finite models), whereas model elimination approaches are more robust for a wider range of applications.

Last but not least, we want to point out that OTTER, SETHEO, and SATCHMO do not support a procedural reading of program clauses—they all need contrapositives and provide little support for computing answers—but PROTEIN does. So, ancestry model elimination provides both an easy reading of problem descriptions as programs and

an efficient processing of them. And that is useful if we want to use logic as a real programming language.

Acknowledgements

We would like to thank François Bry, Jürgen Dix, Bertram Fronhöfer, Ortrun Ibens, Reinhold Letz and William W. McCune for helpful discussions, Bruce Spencer and some anonymous referees for useful comments on this papers, and Olaf Menkens and Dorothea Schäfer for their implementational work.

References

[1] P. Baumgartner, A model elimination calculus with built-in theories, in: H.-J. Ohlbach, ed., *Proceedings of the 16th German AI-Conference (GWAI-92)*, Lecture Notes in Artificial Intelligence **671** (Springer, Berlin, 1992) 30–42.

[2] P. Baumgartner, Refinements of theory model elimination and a variant without contrapositives, *Proceedings ECAI-94*, Amsterdam (1994); Long version: Research Rept. 8/93, Institute for Computer Science, University of Koblenz, Koblenz (1993).

[3] P. Baumgartner and U. Furbach, Consolution as a framework for comparing calculi, *J. Symb. Comput.* **16** (1993).

[4] P. Baumgartner and U. Furbach, Model elimination without contrapositives and its application to PTTP, *J. Autom. Reasoning* **13** (1994) 339–359; Short version in: *Proceedings of CADE-12*, Lecture Notes in Artificial Intelligence **814** (Springer, Berlin, 1994) 87–101.

[5] P. Baumgartner and U. Furbach, PROTEIN: a PROver with a Theory Extension INterface, in: A. Bundy, ed., *Proceedings of the 12th International Conference on Automated Deduction*, Nancy, Lecture Notes in Artificial Intelligence **814** (Springer, Berlin, 1994) 769–773.

[6] P. Baumgartner, U. Furbach and I. Niemelä, Hyper tableaux, in: *JELIA 96. European Workshop on Logic in AI*, Lecture Notes in Artificial Intelligence **1126** (Springer, Berlin, 1996); Long version in: Fachberichte Informatik, 8–96, Universität Koblenz-Landau, Koblenz (1996).

[7] P. Baumgartner and F. Stolzenburg, Constraint model elimination and a PTTP-implementation, in: P. Baumgartner, R. Hähnle and J. Posegga, eds., *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Lecture Notes in Artificial Intelligence **918** (Springer, Berlin, 1995) 201–216.

[8] C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).

[9] R. Demolombe and T. Imielski, *Nonstandard Queries and Nonstamdard Answers* (Oxford University Press, Oxford, 1994).

[10] ECRC GmbH, München, ECLiPSe 3.5: User Manual—Extensions User Manual (1995).

[11] E. Eder, Consolution and its relation with resolution, in: *Proceedings IJCAI-91*, Sydney, Australia (1991).

[12] G. Gottlob and C.G. Fermüller, Removing redundancy from a clause, *Artif. Intell.* **61** (1993) 263–289.

[13] C.C. Green, Theorem-proving by resolution as a basis for question-answering systems, in: B. Meltzer and D. Michie, eds., *Machine Intelligence* **4** (Elsevier, New York, 1969) 183–205.

[14] D. Kapur and P. Narendran, NP-completeness of the set unification and matching problems, in: J.H. Siekmann, ed., *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, Lecture Notes in Artificial Intelligence **230** (Springer, Berlin, 1986) 489–495.

[15] R. Letz, K. Mayr and C. Goller, Controlled integration of the cut rule into connection tableau calculi, *J. Autom. Reasoning* **13** (1994) 297–337.

[16] R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO: a high-performance theorem prover, *J. Autom. Reasoning* **8** (1992).

[17] J. Lloyd, *Foundations of Logic Programming*, Symbolic Computation (Springer, Berlin, 2nd extended ed., 1987).
[18] J. Lobo, J. Minker and A. Rajasekar, *Foundations of Disjunctive Logic Programming* (MIT Press, Cambridge, MA, 1992).
[19] D. Loveland, Mechanical theorem proving by model elimination, *J. ACM* **15** (1968).
[20] D. Loveland, *Automated Theorem Proving—A Logical Basis* (North-Holland, Amsterdam, 1978).
[21] D.W. Loveland, Near-Horn Prolog, in: J.-L. Lassez, ed., *Proceedings Fourth International Conference on Logic Programming*, Melbourne (1987) 456–469.
[22] D. Loveland, Near-Horn Prolog and beyond, *J. Autom. Reasoning* **7** (1991) 1–26.
[23] D.W. Loveland and D.W. Reed, Near-Horn Prolog and the ancestry family of procedures, Tech. Rept. CS-1992-20, Department of Computer Science, Duke University, Durham, NC (1992).
[24] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, in: E. Lusk and R. Overbeek, eds., *Proceedings of the 9th International Conference on Automated Deduction*, Argonne, IL, Lecture Notes in Computer Science **310** (Springer, Berlin, 1988) 415–434.
[25] W.W. McCune, OTTER 3.0 reference manual and guide, Tech. Rept. ANL-94/6, National Laboratory, Argonne, IL (1994).
[26] H.J. Ohlbach, Predicate logic hacker tricks, *J. Autom. Reasoning* **1** (1985) 435–440.
[27] D. Plaisted, Non-Horn clause logic programming without contrapositives, *J. Autom. Reasoning* **4** (1988) 287–325.
[28] U.R. Schmerl, A resolution calculus giving definite answers, Tech. Rept. 9108, Universität der Bundeswehr, München (1991).
[29] M. Schmidt-Schauß, Implication of clauses is undecidable, *Theoret. Comput. Sci.* **59** (1991) 287–296.
[30] Raymond M. Smullyan, *What is the Name of this Book? The Riddle of Dracula and other Logical Puzzles* (Prentice-Hall, Englewood Cliffs, NJ, 1978).
[31] M.E. Stickel, Automated deduction by theory resolution, *J. Autom. Reasoning* **1** (1985) 333–355.
[32] M.E. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler, *J. Autom. Reasoning* **4** (1988) 353–380.
[33] G. Sutcliffe, C. Suttner and T. Yemenis, The TPTP problem library, in: A. Bundy, ed., *Proceedings CADE-12*, Lecture Notes in Artificial Intelligence **814** (Springer, Berlin, 1994).