Model Elimination, Logic Programming and Computing Answers

Peter Baumgartner • Ulrich Furbach • Frieder Stolzenburg
Universitat Koblenz • Institut fiir Informatik
Rheinau 1 • D-56075 Koblenz • Germany
E-mail: {peter,uli,stolzen} @informatik.uni-koblenz.de

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 mechanism for computing answers into the restart variant of ME. 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 (non-trivial) answers to goals, instead of only proving the *existence* of answers.

Keywords. Automated reasoning; theorem proving; model elimination; logic programming; computing answers.

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 much more difficult to compute (non-trivial) answers to goals, instead of only to prove the existence of answers.

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 non-monotonic extensions. However, with respect to interpretation, i.e. proof-theoretic investigations 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 $A\backslash \vee \ldots \vee A_m \longleftarrow B\backslash \wedge \ldots \wedge B_n$ is a representation of the clause $A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$. Indeed, in [Lobo *et al,* 1992] 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 [Lobo *etal,* 1992] completely ignores this aspect by using SLI resolution which requires all contrapositives.

There are proposals for first order proof calculi using program clauses only in this procedural reading, e.g. Plaisted's problem reduction formats [Plaisted, 1988], or the nearHorn-Prolog family introduced by Loveland and his co-workers [Loveland, 1991]. These approaches introduce new calculi or proof procedures, for which efficient implementations still have to be developed. (For a thorough discussion we refer to [Baumgartner and Furbach, 1994a].) Our aim was to modify ME such that it can be used for logic programming in the above sense. This gives us the possibility to use existing theorem provers for disjunctive logic programming. As a first step towards this goal, we introduced in [Baumgartner and Furbach, 1994a] the restart variant of ME and proved its refutational completeness. In this paper, we introduce an answer computing mechanism into restart model elimination (proofs of all stated theorems can be found in the long version [Baumgartner *et al,* 1995]). Furthermore we define a variant called *ancestry restart ME* which allows extended regularity checking (i.e. loop checking) wrt. the ordinary restart ME. Additionally, this variant prefers proofs which allow for definite answers.

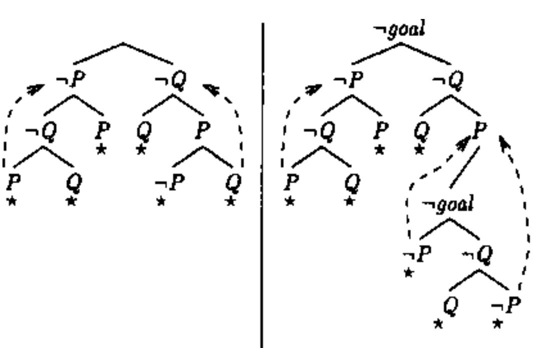
For the second aspect, namely computing answers, we accommodated our PROTEIN system [Baumgartner and Furbach, 1994b] for answer computing as described below. We demonstrate with some of Smullyan's puzzles [Smullyan, 1978] that it is much more difficult to compute answers instead of only to prove unsatisfiability. For this we give a comparative study of high performance theorem provers, including OTTER, SETHEO and our PROTEIN system.

1 From Tableau to Restart Model Elimination

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 a ME calculus that differs from the original one presented in [Loveland, 1968]. It is described in [Letz *et al.,* 1992] as the base for the prover SETHEO. In [Baumgartner and Furbach, 1993] this calculus is discussed in detail by presenting it in a consolution style [Eder, 1991] and compared to various other calculi. ME (in this sense) manipulates trees by extension and reduction steps. In order to recall the calculus consider the clause set

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



A Model Elimination Refutation

A Restart Model Elimination Refutation. Positive goal-nodes are not displayed.

Figure 1: Model Elimination (left side) vs. Restart Model Elimination (right side).

A model elimination refutation is depicted in Figure 1 (left side). It is obtained by successive fanning with clauses from the input set *(extension steps).* Additionally, it is required that every inner node is complementary to one of its sons. Such sons are decorated with a "*" in Figure 1. A dashed arrow indicates a *reduction step,* i.e. the closing of a branch due to a path 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 Figure 1 (left side) the clause {-P, Q} was used to extend the positive leaf P, i.e. we used the program clause *Q* <- P via the body literal *P* and hence did dissent with a procedural reading of the clause.

In the right part of Figure 1 a refutation with the modified version, the *restart ME* calculus, is displayed. 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 *->goal* — is copied, and then an extension follows.

In a variant called *strict* restart model elimination not even reduction steps are allowed at positive leaves. Hence the calculus is forced to apply restart steps wherever possible.

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 [Baumgartner and Furbach, 1993].

1.2 Restart Model Elimination

Instead of trees we now manipulate multisets of paths, where paths are sequences of literals. We will state some basic definitions.

A *clause* is a multiset of literals, usually written as the disjunction $L_1 \vee \ldots \vee L_n$. *A program* is a consistent set of clauses (thus possibly including negative clauses). A *connection* is a pair of literals, written as *(K,L),* which can be made complementary by an application of a substitution. A *path* is a sequence of literals, written as $p = \langle L_1, \ldots, L_n \rangle$. L_n is called the *leaf* of p, which is also denoted by $leaf(p)$; similarly, the first element L_1 is also denoted by $first(p)$. The symbol "$\circ$" denotes the append function for literal sequences.

In the sequel both path sets and sets of literals are always understood as *multi*sets, and usual set notation will be used. Multisets of paths are written with caligraphic capital letters.

From now on we use the notation $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$ as a representation of the clause $A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$. Such clauses are called *program clauses* with *head literals* A_i (if present) and *body literals* B_i.

We assume our clause sets to be in *goal normal form*, i.e. there exists only one goal clause (a clause containing only negative literals), 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 modifying every purely negative clause $\neg B_1 \vee \ldots \vee \neg B_n$ to $goal \leftarrow B_1, \ldots, B_n$.

If $C = A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$ is a clause then its *path set* $\mathcal{P}_C$ is $\{\langle L \rangle \mid L \in \{A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}\}$. The *dot product* $p \cdot \mathcal{Q}$ of a path p and a path set $\mathcal{Q}$ is defined as $\{p \circ q \mid q \in \mathcal{Q}\}$. It can be interpreted as a branching of a path p into the new paths from $\mathcal{Q}$

The inference rule *extension* from the restart ME calculus, will be defined in such a way that one is free in selecting any head literal as part of a connection. For this, we introduce a head selection function.

Definition 1.1 (Head selection Function) A *head selection function* f is a function that maps a clause $A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m$ with $n \geq 1$ 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 $(A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m)\gamma$ (for some substitution γ) then f selects L in $A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m$. **(End Definition)**

Note that this head selection function has nothing to do with the selection function from SLD-resolution which selects subgoals. This will be discussed later.

Definition 1.2 (Strict Restart Model Elimination) Given a set of clauses S and a head selection function.

The inference rule *extension* is defined as follows:

$$\frac{\mathcal{P} \cup \{p\} \quad A_1 \vee \ldots \vee A_i \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n}{\mathcal{R}}$$

where

1. $\mathcal{P} \cup \{p\}$ is a path multiset, and $A_1 \vee \ldots \vee A_i \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$ is a variable disjoint variant of a clause in S; A_i is the selected literal, and
2. $(leaf(p), A_i)$ is a connection with MGU σ, and
3. $\mathcal{R} = (\mathcal{P} \cup \{p \circ \langle K \rangle \mid K \in \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}\})\sigma$

The inference rule *reduction* is defined as follows:

$$\frac{\mathcal{P} \cup \{p\}}{\mathcal{P}\sigma} \quad \text{where}$$

1. $\mathcal{P} \cup \{p\}$ is a path multiset, and
2. there is a positive literal L in p such that $(L, leaf(p))$ is a connection with MGU σ.

The inference rule restart is defined as follows:

$$\frac{\mathcal{P} \cup \{p\}}{\mathcal{P} \cup \{p \circ \langle L \rangle\}} \quad \text{where}$$

1. $\mathcal{P} \cup \{p\}$ is a path multiset, and
2. leaf(p) is a positive literal, and
3. L=first (p).

A strict restart ME derivation from the clause set S consists of a sequence $(\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ and a substitution $\sigma_1 \cdots \sigma_n$, where

1. P_o is a path m u l t $\{\langle L_1\rangle, \ldots, \langle L_n\rangle\}$ i s t i n g of paths of length 1, with L\ $\vee \ldots \vee$ also in S (called the goal clause), and for i = 1 . . . n
2. P_i is obtained from Vi-\ by means of an extension step with an appropriate clause C from S and MGU σ_i, or
3. P_i is obtained from P_i-\ by means of a reduction step and MGU σ_i, or
4. P_i is obtained from P_{i-1} by means of a restart step.

The path p is called selected path in all three inference rules. A restart step followed immediately by an extension step at the just obtained path is also called a restart extension step. Finally, a refutation is a derivation where V_n = {}. (End Definition)

Note that in extension steps we can connect only with the head literals of input clauses. Since in general this restriction is too strong, we have to "restart" the computation with a fresh copy of a negative clause. This is achieved by the restart rule, because refutations of programs in goal normal form always start with -goal, i.e. the copied literal first(p) = -goal; furthermore, only extension steps are possible to -goal, introducing a new copy of a negative clause (cf. Figure 1, right side).

The reduction operation is permitted from negative leaf literals to positive ancestor literals only. This condition can be relaxed towards disregarding the sign, which then yields the non-strict calculus version. See [Baumgartner and Furbach, 1994a] for a discussion of the differences. The reader aware of this work will notice that in the present text we define the calculus slightly different. This happens in order to conveniently express another calculus variant defined below.

Note that the restart ME calculus does not assume a special selection function which determines which path is to be extended or reduced next. Correctness and completeness of this calculus follows immediately from a result in [Baumgartner, 1994]. From the definition of the inference rule extension, it follows immediately, that this calculus only needs those contrapositives of clauses which contain a positive literal in their heads.

2 Computing Answers

In this section we introduce the notion of computed answers and we state an answer completeness result for restart ME. We assume as given a program P together with one single query $\leftarrow G_1 \wedge \ldots \wedge G_n$, where the G_iS are positive literals. We will often abbreviate such a query as <— Q, where Q stands for the conjunction of G_iS. The clause set S is the transformation of $P \cup \{\leftarrow Q\}$ into goal normal form. In the following definition of computed answer we collect applications of the query clause, but not applications of negative clauses from the program P.

Definition 2.1 (Answers) If <- Q is a query and $\theta_1, \ldots, \theta_m$ are substitutions for the variables from Q. then $Q\theta_1 \vee \ldots \vee Q\theta_m$ is an answer (for S). An answer $Q\theta_1 \vee \ldots \vee Q\theta_m$ is a correct answer if $P \models \forall(Q\theta_1 \vee \ldots \vee Q\theta_m)$. Let now a restart ME refutation of S with goal clause <- goal and substitution a 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 +- $Q\rho_i$ where ρ_i, is the renaming substitution of Let $\sigma_i = \rho_i\sigma|_{dom(\rho_i)}$. Then $Q\sigma_1 \vee \ldots \vee Q\sigma_m$ is a computed answer (for S). (End Definition)

Theorem 2.2 (Lifting Theorem for Restart Model Elimination) Let S' be a set of ground instances of clauses taken from a clause set S. Assume there exists a restart ME derivation $D' \equiv P'_0, P'_1, \ldots, P'_n$ *from* S' with goal clause C_0 € S'. Then there exists a restart ME derivation D = P_o, P_1,…, P_n from S with some goal $C_0 \in S$ and substitution o such that P_n is more general than P_n. (A path set P is more general than a path set Q iff for some substitution 6 we have PS = Q.)

Furthermore, there exists a substitution δ such that P_i is obtained from P'_{i-1} by an extension step with clause C E S' if and only if P_i is obtained from P_{i-1} by an extension step with a clause $C \in S$ such that $C\rho\sigma\delta = C'$, where p is the renaming substitution applied in that extension step.

The first part of the theorem will be used in the proof of refutational completeness (because for a refutation on the ground level, i.e. a derivation of P'_n = {}, only the empty path set P_n — {} can be more general), while the second part will be used in the proof of answer completeness (Theorem 2.3). In particular, to obtain this we have to demand one single substitution 6 which maps any of the clauses $C\rho\sigma$ 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 [Lobo et al., 1992] 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.

Theorem 2.3 (Answer completeness of restart ME) // $Q\theta_1 \vee \ldots \vee Q\theta_l$ is a correct answer for a program P, then there exists a strict restart ME refutation from S with computed answer $Q\sigma_1 \vee \ldots \vee Q\sigma_m$ such that $Q\sigma_1 \vee \ldots \vee Q\sigma_m$ entails $Q\theta_1 \vee \ldots \vee Q\theta_l$, *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 6 to a subclause of the given answer (and hence implies it). Unfortunately we can not obtain a result stating that the computed answer contains less (or equal) literals than the given answer.

All proofs are stated in the long version of this paper [Baumgartner et al, 1995].

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 path. Expressed more semantically, it says that it is never necessary to repeat in a derivation a previously derived subgoal (viewing open leaves as subgoals).

Unfortunately, regularity is *not* compatible to restart ME. In this section we will present a variant of restart ME, the *ancestry restart* variant, which allows for extended regularity checks. This variant is motivated by Loveland's UnH-Prolog [Loveland and Reed, 1992].

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.

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 restart ME is modified in such a way 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 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 3.1 (Ancestry Restart Model Elimination) The calculus *ancestry restart ME* is the same as *strict restart ME* (Definition 1.2), except that the inference rule *restart* is modified by replacing the condition 3. by the new condition 3'.:

3'. L is a negative literal occurring in p. In this context L is also called the *restart literal.*

The modified rule is called *ancestry restart.* (End Definition)

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 can be simulated in ancestry restart ME by a restart step followed by a strict reduction step. Thus, non-strictness is "built-in" into ancestry restart ME.

Note that the ancestry restart rule includes the restart rule since the first literal can be used for the restart as well.

Clearly, in terms of a proof procedure the ancestry restart rule induces a larger local search space than the restart rule. On the other side, refutations may become much shorter. 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.

Now we are going towards an appropriate completeness result wrt. 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 3.2 (Regularity) Let p be path written as follows (the As and Bs are atoms):

$$p = \neg B^1_1 \cdots \neg B^1_{k_1} A^1 \neg B^2_1 \cdots \neg B^2_{k_2} A^2 \cdots A^{n-1} \neg B^n_1 \cdots \neg B^n_{k_n}$$

Then p is called *blockwise regular* iff

1. $A^i \neq A^j$ for $1 \leq i,j \leq n-1$, $i \neq j$ *(Regularity wrt. positive literals)* and
2. $B^l_i \neq B^l_j$ for $1 \leq l \leq n$, $1 \leq i,j \leq k_l$, $i \neq j$ *(Regularity inside blocks).*

If additionally it holds that

3. $B^l_i \neq B^m_j$ for $1 \leq l < m \leq n$, $1 \leq i \leq k_l$, $2 \leq j \leq k_m$ *(Global negative regularity)*

then p is called *globally regular.* A path set is called *(blockwise, globally) regular* iff every path in it is (blockwise, globally) regular. Similarly, a derivation is called *(blockwise, globally) regular* iff every of its path sets is (blockwise, globally) regular. (End Definition)

Condition 1 states that all positive literals along a path are pairwise different, and condition 2 states that negative literals inside blocks are pairwise different, where by a block we mean a smallest subpath delimited by positive literals or the ends of the path. Condition 3 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 Figure 1 (right side) is not globally regular, as can be seen by the two occurrences of -Q in the rightmost path. From this example we learn that restart ME is incompatible with the global regularity restriction. However it holds:

Theorem 3.3 (Completeness of Ancestry Restart Model Elimination) *Let f be a head selection function and S be an unsatisfiable clause set in goal-normal form. Then there exists a globally regular ancestry restart ME refutation ofS starting with <— goal and selection function f.*

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

Theorem 3.4 (Answer completeness of ancestry restart ME) *Ancestry restart ME is answer complete in the sense of Theorem 2.3. In particula $Q\theta$ is a correct definite answer for a program P, then there exists an ancestry restart ME refutation from P with computed ans $Q\sigma$ such that $Q\sigma\delta = Q\theta$, for some substituti δ Furthermore, the input clause goal <— Q is used exactly once, namely at the first extension step of <— goal.*

The last theorem enables us to enumerate definite answers *only,* by simply restricting the use of *goal <— 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.

4 Implementation

All variants and refinements of ME discussed so far, i.e. the restart, strict and ancestry variants (possibly with selection function), loop checking by regularity and factorization, are implemented in the PROTEIN system [Baumgartner and Furbach, 1994b]. It is a first order theorem prover based on the Prolog technology theorem proving (PTTP) technique, implemented in ECLiPSe-Prolog.

Since ME is a goal-oriented, linear and answer complete calculus, it is well suited as an interpreter for disjunctive logic programming. PROTEIN facilitates computing disjunctive and definite answers. In its newest release their is also a flag which allows us to look for definite answers only.

5 Comparative Theorem Prover Study

In the sequel, we want to tell about 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 besides "yes" or (possibly) "no". - We will illustrate our experiences with a puzzle example which allows for indefinite and definite answers.

5.1 Knights and Knaves

The example follows problem #36 in [Smullyan, 1978]. A similar example is studied in [Ohlbach, 1985]. The natural language description of the problem is stated below. There, the last two pieces of information 5 and 6 explicitly state some knowledge about inferencing. We need them in order to be able to cope with the information in 2 because our description language is first order.

> 1. On an island, there live exactly two types of people: knights and knaves. 2. Knights always tell the truth and knaves always lie. 3.1 landed on the island, met two inhabitants, asked one of them: "Is one of you a knight?" and he answered me. 4. What can be said about the types of the asked and the other person depending on the answer I get? - 5. We assume, that either a proposition or its negation is true. 6. If the disjunction of two propositions is true then at least one of them must be true.

In our formalization of the problem below, the formulae in 1 and 2 express the corresponding pieces of information from above. Depending on the case considered, we choose one of the formulae (a) or (b) in 3. 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 4 can be considered as the query. We have to express the pieces of information 5 and 6 explicitly by introducing the unary predicate true. The transformation of the formulae below into clausal form is straightforward and therefore omitted here. It consists of 11 clauses. - The symbol $\veebar$ denotes exclusive or.

1. $true(isa(Q, knight)) \veebar true(isa(Q, knave))$
2. $says(P, S) \rightarrow (true(S) \leftrightarrow true(isa(P, knight)))$
3. (a) $says(asked, \bullet)$ ("yes")
   (b) $says(asked, not(\bullet))$ ("no")
   where $\bullet = or(isa(asked, knight), isa(other, knight))$
4. $\neg true(isa(asked, X)) \vee \neg true(isa(other, Y))$
5. $true(not(C)) \veebar true(C)$
6. $true(or(A, B)) \leftrightarrow (true(A) \vee true(B))$

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 1 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) there exists a definite answer. It follows a list of these possible answers where X / Y is an abbreviation of true(isa(asked X)) $\wedge$ true(isa(other, Y)).

1. knave/knave V knave/knight V knight/knave V knight / knight (trivial)
2. knave /knave V knight /knave V knight / knight (indefinite)
3. knave /knight (definite)

Before turning to our experiments we want to mention some interesting facts. Firstly, answer completeness requires that we are able to compute the indefinite and definite answer in the respective cases. Secondly, to derive these answers we need a clause set which is not minimal unsatisfiable; notice that the clauses of 1 and 4 together are (minimal) unsatisfiable yielding the trivial answer. Thirdly, 9 extension steps are needed to derive the indefinite or the definite answer respectively, while only 7 extension steps are needed to derive the trivial answer (in both cases). - These remarks indicate that it should be more difficult to find the more precise answers.

5.2 Experimental Results

We tried to get the answers from above automatically by using the theorem proving systems OTTER [McCune, 1994] which is a resolution-style theorem proving program coded in C for first order logic (with equality), SETHEO [Letz et a/., 1992] which is a top-down prover for first order predicate logic based on the calculus of the so-called connection tableaux which generalizes weak ME, implemented in C, and PROTEIN [Baumgartner and Furbach, 1994b] which we already introduced in Section 4. - We used the clause ordering given by the problem description, but our experiments show that the (run time) results depend on the ordering.

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 (a) where the search stops after finding 15 times only the trivial answer with binary resolution. However, we find a proof by using hyper-resolution with factorization immediately within 0.4s. - There is a solution to the problem with subsumption; it can be shown that we only have to take the answer literals into account during the subsumption steps. 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 within 1.0s and the definite answer within 0.6s. That is quite good and may be explained by the subgoal reordering heuristics built into SETHEO, which are not (yet) incorporated into our system. But in addition, SETHEO also has subsumption constraints which are used in the default setting. It is not quite clear, whether these constraints destroy answer completeness in SETHEO. - Table 2 shows the timings for OTTER and SETHEO. All timings are measured on a Sparc 10. The symbol oo denotes the fact that no proof was found within 1 hour; that is true for OTTER applied to case (b) of our example.

Prover	Answer	Time (s)	Settings
OTTER	trivial	2.1	plain hyper-resolution
	indefinite	0.4	hyper-resolution + factor.
	definite	∞	several trials
SETHEO	trivial	0.5	with constraints
	indefinite	1.0	with constraints
	definite	0.6	with constraints
PROTEIN	trivial	0.5	any setting
	indefinite	∞	plain ME
		41.4	restart + sel. function
	definite	2022.8	plain ME
		38.4	ancestry restart

Figure 2: Timings

PROTEIN *is* answer complete; that has been stated in this paper. It finds out the indefinite and definite answer for the respective case. The table in Figure 2 also shows some timings for finding these answers with PROTEIN. We tried both, plain and restart ME. In 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. By this, we get a significant speed up of the search. - As one can see, using restart helps for this problem, since plain ME does not find the desired answers quickly, although it does so for trivial answers. But it is not quite clear which flags should be used in addition.

We investigated more puzzle examples from [Smullyan, 1978]. All our experiments corroborate the following facts: resolution has difficulties in solving puzzles because of the problem with subsumption; model elimination is better suited although it could not solve all puzzles that we tested. For example, OTTER needs 281.8s on puzzle #35 while PROTEIN only needs 153.1s. Further investigations are necessary. It seems that also a model generation approach is very adequate [Manthey and Bry, 1988] for these kind of problems because they often allow for finite models. Last but not least, we want to point out that both, OTTER and SETHEO do not support a procedural reading of program clauses - they need all contrapositives - but PROTEIN does; and that is useful if we want to use logic as a real programming language.

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 out yet more efficient calculi and to incorporate nonmonotonic extensions.

Acknowledgements

We would like to thank Francois Bry, Jurgen Dix, Bertram Fronhofer, Reinhold Letz and William W McCune for helpful discussions, and Olaf Menkens and Dorothea Schafer for their implementational work.

References

[Baumgartner and Furbach, 1993] P. Baumgartner and U. Furbach. Consolution as a Framework for Comparing Calculi. *Journal of Symbolic Computation,* 16(5), 1993. Academic Press.

[Baumgartner and Furbach, 1994a] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives and its Application to PTTP. *Journal of Automated Reasoning,* 13:339-359,1994. Short version in: Proceedings of CADE-12, Springer LNAI 814, 1994, pp 87-101.

[Baumgartner and Furbach, 1994b] P. Baumgartner and U. Furbach. PROTEIN: A *PROver* with a Theory Extension /nterface. In A. Bundy, editor, *Automated Deduction - CADE-12,* volume 814 of *LNAI,* pages 769-773. Springer, 1994.

[Baumgartner *et al,* 1995] Peter Baumgartner, Ulrich Furbach, and Frieder Stolzenburg. Model elimination, logic programming and computing answers. Fachberichte Informatik 1/95, Universitat Koblenz-Landau, Koblenz, 1995.

[Baumgartner, 1994] P. Baumgartner. Refinements of Theory Model Elimination and a Variant without Contrapositives. In A.G. Cohn, editor, *11th European Conference on Artificial Intelligence ECAI94.* Wiley, 1994. (Long version in: Research Report 8/93, University of Koblenz, Institute for Computer Science, Koblenz, Germany).

[Eder, 1991] E. Eder. Consolution and its Relation with Resolution. *In Proc.IJCAI'91,* 1991.

[Letz *etal,* 1992] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning,* 8(2), 1992.

[Lobo *et ai,* 1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, Cambridge, MA, London, England, 1992.

[Loveland and Reed, 1992] D. Loveland and D. Reed. Near-Horn Prolog and the Ancestry Family of Procedures. Technical Report CS-1992-20, Department of Computer Science, Duke University, Durham, North Carolina, December 1992.

[Loveland, 1968] D. Loveland. Mechanical Theorem Proving by Model Elimination. *JACM,* 15(2), 1968.

[Loveland, 1991] D. Loveland. Near-Horn Prolog and Beyond. *Journal of Automated Reasoning,* 7:1-26, 1991.

[Manthey and Bry, 1988] Rainer Manthey and Francois Bry. SATCHMO: a theorem prover implemented in Prolog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction, Argonne, Illino USA, May 1988,* pages 415-434. Springer, Berlin, Heidelberg, New York, 1988. LNCS 310.

[McCune, 1994] William W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, National Laboratory, Argonne, IL, January 1994.

[Ohlbach, 1985] Hans Jurgen Ohlbach. Predicate logic hacker tricks. *Journal of Automated Reasoning,* 1:435-440, 1985.

[Plaisted, 1988] D. Plaisted. Non-Horn Clause Logic Programming Without Contrapositives. *Journal of Automated Reasoning,* 4:287-325, 1988.

[Smullyan, 1978] Raymond M. Smullyan. *What is the name of this book? The riddle of Dracula and other logical puzzles.* Prentice-Hall, Englewood Cliffs, NJ, 1978.