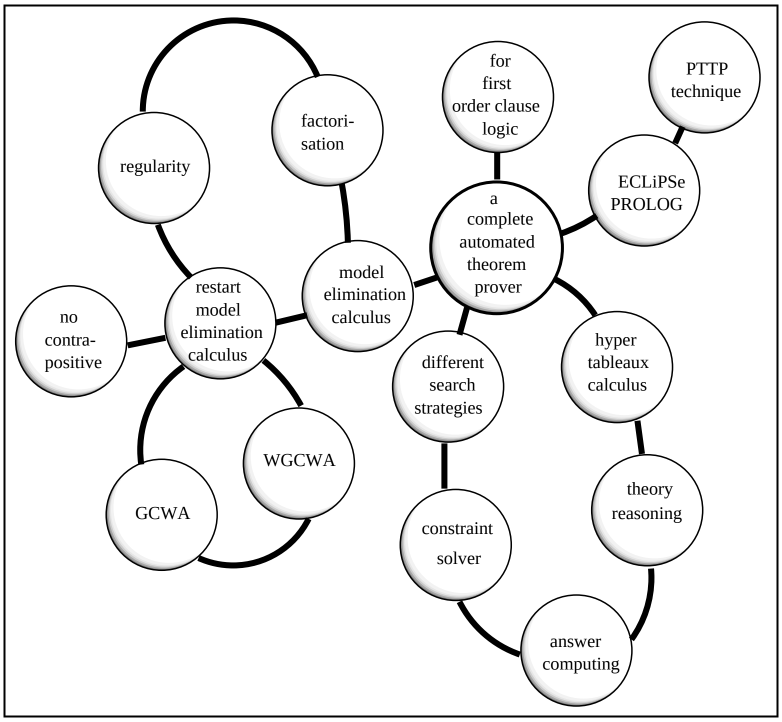
PROTEIN
A **PRO**ver with a **T**heory **E**xtension **IN**terface

Dorothea Schäfer
E-mail: `doro@informatik.uni-koblenz.de`

November 8, 1996



Contents

List of Figures

List of Tables

1 Introduction

PROTEIN is a complete automated theorem prover for first order clause logic offering the following features:

- PROTEIN follows *Loveland*'s model elimination procedure ([Lov68], $\hookrightarrow$ Section 2.1) and is implemented according to *Stickel*'s PTTP (**P**rolog **T**echnology **T**heorem **P**rover) technique ([Sti88]), i.e. the input specification is compiled into a prolog program ($\hookrightarrow$ Section 4.1-4.2).
- PROTEIN masters a modification of model elimination called restart model elimination ($\hookrightarrow$ Section 2.2) that does not need all contrapositives ([BauFur94b], $\hookrightarrow$ Section 2.2). Different versions of restart model elimination are implemented which are strict restart model elimination, ancestry restart model elimination, and restart model elimination with selection function that avoids all contrapositives ($\hookrightarrow$ Section 4.1-4.3). An implementation variant of restart model elimination is delayed restart model elimination which enables PROTEIN to compute saturated tableaux ($\hookrightarrow$ Section 4.3).
- PROTEIN covers several improvements like ground reduction steps ($\hookrightarrow$ Section 4.2), regularity ($\hookrightarrow$ Section 4.4), and factorisation ($\hookrightarrow$ Section 4.5).
- PROTEIN includes theory reasoning which relieves a calculus 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 ([Bau93a], $\hookrightarrow$ Section 2.3). Several refinements of theory reasoning have been implemented ($\hookrightarrow$ Section 4.6).
- PROTEIN offers different search strategies like depth first search, iterative tree deepening, and iterative term deepening ($\hookrightarrow$ Section 4.8).
- PROTEIN offers an answer predicate to compute answer substitutions ($\hookrightarrow$ Section 4.9). All modifications of model elimination implemented in PROTEIN are answer complete ([BauFurSto95]).
- PROTEIN's latest features are a hyper tableaux calculus ($\hookrightarrow$ Section 2.4).
- PROTEIN was developed at the *University of Koblenz* ([URL-Uni]) in the course of the *DFG*-project "Deduktion".

2 Theoretical Background

In this section the theoretical background of PROTEIN is introduced. The starting point are some definitions to be prepared to define the basic calculus of PROTEIN, model elimination, which is a calculus for first order clause logic. Then a variant called restart model elimination and several modifications of it are presented.

2.1 Model Elimination

The following definition of model elimination differs from the original, introduced by *Loveland* ([Lov68]) in 1968. It follows [BauFur94a] where model elimination manipulates trees which are described as multisets of paths.

First some basic definitions:

Definition 2.1 (clause)
A clause is a multiset of literals $\{L_1, \ldots, L_k\}$ usually written as the disjunction $L_1 \vee \ldots \vee L_k$. $\Box$

As usual, clauses are considered implicitly as being universally quantified, and a clause set is considered logically as a conjunction of clauses.

In the following a clause $C = \{L_1, \ldots, L_k\}$ is also written in the logic programming style as $A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$, $n, m \geq 0$, $n + m = k$, A_i positive literals of C, B_j negated negative literals of C, to separate the positive and the negative literals of a clause. This form is also used in the input files of PROTEIN.

Definition 2.2 (path)
A path is a sequence of literals written as $p = \langle L_1, \ldots, L_n \rangle$. $\Box$

L_n is called the leaf which is also indicated by $leaf(p)$. The first literal is also denoted by $first(p)$. The symbol $\circ$ stands for the append function of sequences, the symbol $\dot{\in}$ for element of a sequence. In the following multisets of paths are written with calligraphic capital letters. Paths may be selected from the multiset by a computation rule which is a function from the path multiset to a element of it. In [Bau93a] it is proven that any computation rule is allowed for model elimination derivations.

Definition 2.3 (connection)
A connection (L_1, L_2) is a pair of literals which can be made complementary by application of a substitution. $\Box$

We assume our clause sets to be in a normal form which is similar to the **goal normal form**, i.e. there exists one goal, 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 $\leftarrow query$ where $query$ is a new predicate symbol, and by adding $query \leftarrow B_1, \ldots, B_n$ for every purely negative clause $\leftarrow B_1, \ldots, B_n$. This transformation preserves both soundness and completeness. All derivations start with $\leftarrow query$ whatever clause set is to prove. In the originally definition of the goal normal form the purely negative clauses are replaced by the so called query clauses, which is not complete for model elimination.

Example 2.4 (all-for-two) Consider the clause set

$$\begin{array}{l} p, q \leftarrow \\ q \leftarrow p \\ p \leftarrow q \\ \leftarrow p, q \end{array}$$

transformed into normal form for model elimination

$$\begin{array}{l} \leftarrow query \\ query \leftarrow p, q \\ p, q \leftarrow \\ q \leftarrow p \\ p \leftarrow q \\ \leftarrow p, q \end{array}$$

p, q, $query$ are atoms □

Now, we are prepared for the definition of model elimination which consists of two inference rules dealing with a given set of clauses M and a given computation rule.

Definition 2.5 (model elimination)

- The **inference rule extension** is defined as follows:

  $$\frac{\mathcal{P} \cup \{p\} \quad C}{(\mathcal{P} \cup \{p \circ \langle L_i \rangle \,|\, L_i \in C \setminus \{L\}\})\sigma} \quad \text{where}$$

  1. $\mathcal{P} \cup \{p\}$ is a multiset of paths with a selected path p
  2. C is a variable disjoint variant of a clause in M
  3. there is a literal $L \in C$ with $(L, leaf(p))$ is a connection with MGU[1] σ

- 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 multiset of paths with a selected path p
  2. there is a literal $L \dot{\in} p$ with $(L, leaf(p))$ is a connection with MGU σ

- A sequence $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$ is called a **model elimination derivation** iff

  1. $\mathcal{P}_1 = \{\langle \neg query \rangle\}$ using the normal form transformation of clause sets from page 5

[1] most general unifier

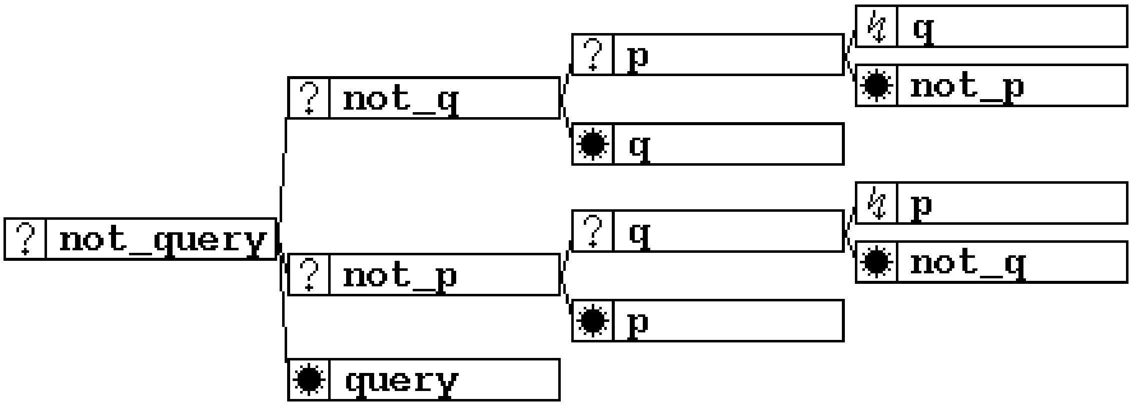


Figure 1: a model elimination refutation for all-for-two

2. $\mathcal{P}_{i+1}$ is obtained from $\mathcal{P}_i$ by means of an extension step with an appropriate C or from $\mathcal{P}_i$ by means of a reduction step

- A **model elimination refutation** is a derivation where $\mathcal{P}_n = \emptyset$

□

A lifting theorem for model elimination is given in [BauFurSto95]. A model elimination refutation of Example 2.4 is depicted in a graphical tree notation in Figure 1. The tree is created automatically by PROTEIN and printed on screen by a module called `tview` (see Section 4.10). The proof starts with $\{\langle\neg query\rangle\}$ followed immediately by an extension step with the clause $query \leftarrow p, q$. An extension step is performed by selecting one path of the tree and an input clause that contains a literal which is complementary by unification with the leaf of that path. The selected path is extended with the literals of the clause as new leafs and replaced in the path multiset by the new paths except the one containing the connection. This one is closed which is marked with a sun. The other leafs are marked with question marks. The paths are fanned successively with clauses from the input set by extension steps until reduction steps can be applied to close paths. Whenever a leaf of a path has a connection to one of its ancestor literals, the path may be closed, deleted from the path multiset, which is indicated by a flash. Extension and reduction steps are allowed at any leaf of the tree and for extensions any literal from an input clause can be used to form a complementary pair of literals. Every path of the tree in Figure 1 is closed, there are no leafs with question marks, so the path set is empty which shows that this tree is a graphical representation of a refutation of the clause set in Example 2.4. Notice that in a refutation tree every inner node is complementary to one of its sons.

2.2 Restart Model Elimination

The only difference between model elimination and restart model elimination 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 $\neg query$ is copied and then an extension follows. In PROTEIN $\neg query$ for restart steps is named $\neg restart$ to distinguish between the starting point of a proof and restart steps. Therefore beside query clauses[2] additionally restart clauses[3] are generated during the normal form transformation. Purely negative clauses are now replaced by query and restart clauses.

Example 2.6 (all-for-two) The clause set from Example 2.4 is transformed into normal form for restart model elimination

$$
\begin{array}{l}
\leftarrow query \\
query \leftarrow p, q \\
restart \leftarrow p, q \\
p, q \leftarrow \\
q \leftarrow p \\
p \leftarrow q
\end{array}
$$

p, q, $query$, $restart$ are atoms □

This section presents a formal definition of the calculus with help of Definition 2.5 of model elimination. The inference rule extension is modified such that no extension step can be applied to a positive leaf. Second, the inference rule restart is defined.

Definition 2.7 (restart model elimination)

- For the **inference rule extension** replace condition 3 of the extension rule in Definition 2.5 by:

  3. there is a *positive* literal $L \in C$ with $(L, leaf(p))$ is a connection with MGU σ

- The **inference rule reduction** is taken from Definition 2.5

- 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 multiset of paths with a selected path p
  2. $leaf(p)$ is a positive literal
  3. $L = first(p) = \neg query$[4]

- For the definition of a **restart model elimination derivation** replace condition 2 of a model elimination derivation in Definition 2.5 by:

  2. $\mathcal{P}_{i+1}$ is obtained from $\mathcal{P}_i$ by means of an extension step with an appropriate C or from $\mathcal{P}_i$ by means of a reduction or a restart step

[2]clauses of form $query \leftarrow B_1, \ldots, B_n$
[3]clauses of form $restart \leftarrow B_1, \ldots, B_n$
[4]$L = \neg restart$ in PROTEIN

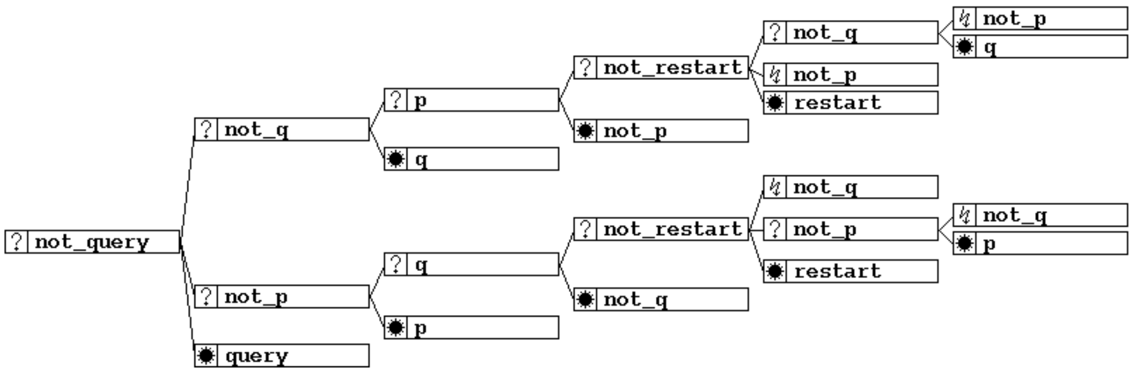


Figure 2: a restart model elimination refutation for all-for-two

- The definition of a **restart model elimination refutation** is taken from Definition 2.5

□

In [BauFurSto95] you find the lifting theorem for restart model elimination. For illustration of the calculus see Figure 2 which shows a restart model elimination refutation of Example 2.4. Instead of extension steps at positive leafs restart steps are carried out. Note, that in extension steps we can connect only with the positive literals of input clauses. Since in general this is a too strong restriction we have to "restart" the computation with a fresh copy of negative clauses. This is achieved by the restart rule.

In the following subsections three modifications of restart model elimination

- strict restart model elimination
- restart model elimination with selection function
- ancestry restart model elimination

are presented, which can be combined to form several variants of restart model elimination.

2.2.1 Strict Restart Model Elimination

In this modification of restart model elimination not even reduction steps are allowed at positive leafs. Hence, the calculus is forced to apply restart steps wherever possible.

Definition 2.8 (strict restart model elimination)

- For the **inference rule reduction** replace condition 2 of the reduction rule in Definition 2.5 by:

  2. there is a *positive* literal $L \dot{\in} p$ with $(L, leaf(p))$ is a connection with MGU σ

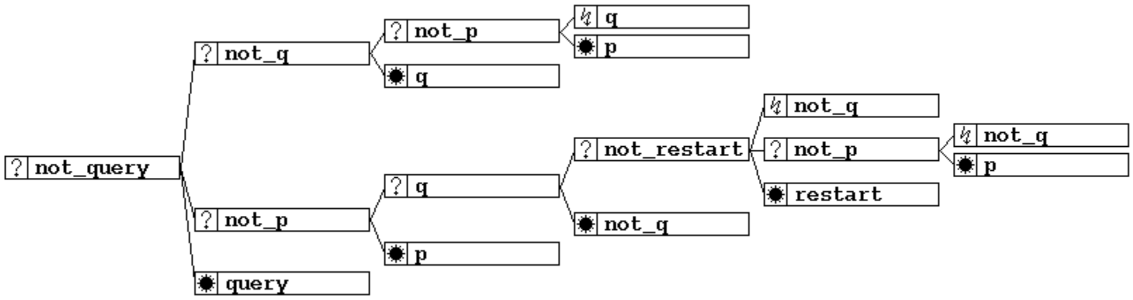


Figure 3: a rme refutation with selection function for all-for-two

- The definitions of the **inference rule extension**, the **inference rule restart**, a **strict restart model elimination derivation**, and a **strict restart model elimination refutation** are taken from Definition 2.7

□

In [BauFur94b] the ground completeness proof for this variant is given. The derivation tree in Figure 2 is strict. The reduction operation (indication flash) is applied from negative leaf literals to positive ancestor literals only.

2.2.2 Restart Model Elimination with Selection Function

In the next modification of restart model elimination which forms the smallest local search space, the inference rule extension is modified such that only one head literal may be used to form a connection. For this we introduce a head selection function.

Definition 2.9 (head selection function)
A head selection function f maps a clause $A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$, $n \geq 1$ to a literal $L \in \{A_1, \ldots, A_n\}$. □

L is called the selected literal of that clause by f.

Definition 2.10 (restart model elimination with selection function)

- For the **inference rule extension** replace condition 3 of the extension rule in Definition 2.5 by:

  3. there is a selected literal $L \in C$ with $(L, leaf(p))$ is a connection with MGU σ

- The definitions of the **inference rule reduction**, the **inference rule restart**, a **restart model elimination with selection function derivation**, and a **restart model elimination with selection function refutation** are taken from Definition 2.7

□

Figure 3 shows a refutation with selection function. Only the first positive literal of a clause may be used to build a connection. That is why $\neg q$ of the topmost path is

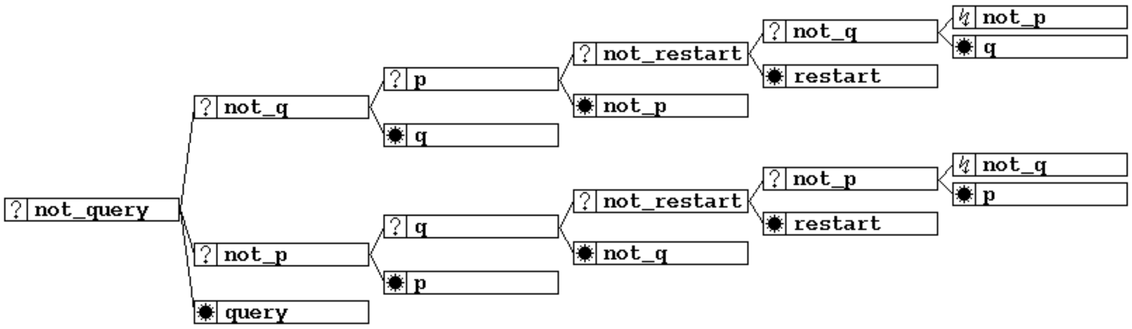


Figure 4: an ancestry restart model elimination refutation for all-for-two

not extended with clause $p, q \leftarrow$ like in Figure 2. A reduction step is applied to the positive literal q, so it is a non strict refutation tree of the clause set in Example 2.4.

2.2.3 Ancestry Restart Model Elimination

Finally, restart model elimination is modified in such a way that every negative literal along a path, not only the topmost literal, may be used for a restart step.

Definition 2.11 (ancestry restart model elimination)

- For the **inference rule restart** replace condition 3 in Definition 2.7 by:

  3. $L \dot{\in} p$ and L negative Literal

- The definition of the **inference rule extension**, the **inference rule reduction**, an **ancestry restart model elimination derivation**, and an **ancestry restart model elimination refutation** are taken from 2.7

□

[BauFurSto95] gives the answer completeness proof of the ancestry variant. The term "ancestry" in the definition is explained by the use of ancestor literals for restart steps. In Figure 4 you see the usage of an ancestry literal instead of the query clause to extend a restart leaf.

Note, that any reduction step from a positive leaf literal to a negative ancestor literal can be simulated in ancestry restart model elimination by a restart step followed by a strict reduction step. Thus, non-strictness is "built-in" into ancestry restart model elimination. The ancestry restart rule includes the restart rule since the first literal can be used for the restart as well.

2.3 Theory Model Elimination

Theory reasoning was introduced by *Stickel* within the resolution calculus ([Sti83], [Sti85]), for model elimination it is defined and investigated in [Bau92] and [Bau93a]. Theory reasoning is described as to relieve a calculus from explicit reasoning in some domains (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 specialised "background" reasoner for theory reasoning.

Theory reasoning comes in two variants: total and partial theory reasoning. The total variant lifts the idea of finding syntactical complementary literals in inferences to a semantic level. Partial theory reasoning tries to break the "big" total step into manageable decidable smaller steps. For a discussion of the two methods see [Bau93a].

The definitions follow [Bau96].

Definition 2.12 (theory inference rule)
A theory inference rule is a pair $P \to C$, where P is a nonempty multiset of literals and C is either a nonempty set of literals or the singleton $\{false\}$. □

P is called premise and C is called conclusio of the inference rule. We assume a theory to be given by a satisfiable set of inference rules. They are stated in the theory environment of an input file of PROTEIN, for syntax see Section 3.1.5. In the sequel $\mathcal{T}$ always denotes such a theory.

Definition 2.13 (minimal first-order theory inference)
We say a clause $R\sigma$ is inferred from a literal multiset K by means of a minimal first-order theory inference with inference rule $P \to C$ and substitution σ, written as $K \Rightarrow_{P \to C,\sigma} R\sigma$ iff

1. $K\sigma = P\sigma$ with MGU σ
2. $R = \begin{cases} \emptyset & \text{if } C = false \\ C & \text{else} \end{cases}$

□

K is called key set and R residue of the theory inference.

Definition 2.14 (inference rule theory extension)
For a given theory $\mathcal{T}$ and set of clauses M the theory extension inference rule is defined as follows:

$$\frac{\mathcal{P} \cup \{p\} \quad L_1 \vee R_1 \quad \ldots \quad L_n \vee R_n}{(\mathcal{P} \cup \{p \circ \langle L \rangle \,|L \in R \vee R_1 \vee \ldots \vee R_n\})\sigma} \quad \text{where}$$

1. $\mathcal{P} \cup \{p\}$ is a multiset of paths with a selected path p
2. $L_i \vee R_i$ are variable disjoint variants of clauses in M with L_is extending literals and R_is the rest of the clauses, $n \geq 0$
3. $K \Rightarrow_{P \to C,\sigma} R\sigma$
4. $K = B \cup \{leaf(p)\} \cup \{L_1, \ldots, L_n\}$ for some $B \subseteq \{L | L \dot{\in} p\}$

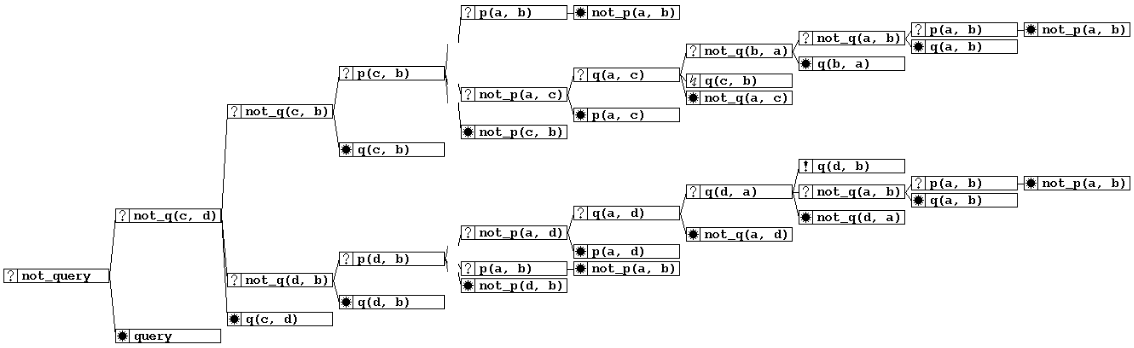


Figure 5: a theory model elimination refutation of MSC006

5. $P \rightarrow C$ is a variable disjoint variant of a theory inference rule from $\mathcal{T}$

$\square$

A theory extension step with $R = \emptyset$ is called total otherwise it is partial. K is called the key set of the extension step.

Extension steps derive from a multiset of paths and some clauses a new multiset of paths or remove paths ending in $false$. In Figure 5 for illustratuion you see nodes which are extending literals, but in general they are not contained in paths in the implementation (see Section 4.6). The theory extension rule is added to the inference rules of a variant of model elimination to receive a theory model elimination calculus variant.

To transform a Horn theory into an efficient background reasoner the method of linearised completion is used which is introduced in [Bau93b]. Additionally a module to detect theories within the input files of PROTEIN has been implemented. It scans through a given input file for PROTEIN to find specified axioms which then will be completed and automatically included in the input file. Both modules are available via [URL-PROTEIN].

Example 5.2 contains an ordinary clause set without theory inference rules for the background reasoner. With help of the theory scanner and the linearised completion tool the clause set from Example 5.3 is constructed which contains the mentioned division of clauses. A refutation of this Example is given in Figure 5.

2.4 Hyper Tableaux

Finally, the calculus hyper tableaux is introduced which is no variant of a model elimination calculus. This calculus is presented and discussed in [BauFurNie96]. Hyper tableaux combines the idea from hyper resolution and from analytic tableaux and can be imitated by theory restart model elimination. How this emulation works is described in Section 4.7.

For a given set of clauses M and a computation rule the hyper tableaux calculus is defined as follows:

Definition 2.15 (hyper tableaux)

- The **inference rule hyper extension** is defined as follows:

  $$\frac{\mathcal{P} \cup \{p\} \quad C}{(\mathcal{P} \cup \{p \circ \langle A_i \rangle \,|0 \leq i \leq n\})\sigma} \quad \text{where}$$

  1. $\mathcal{P} \cup \{p\}$ is a multiset of paths with a selected path p
  2. $C = A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$ is a variable disjoint variant of a clause in M
  3. for all $0 \leq j \leq m$ there is a connection (B_j, L_j) with $L_j \dot{\in} p$ and simultaneous MGU σ.

- A sequence $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$ is called a **hyper tableaux derivation** iff

  1. $\mathcal{P}_1 = \{\langle A \rangle \,|A \in C\}$, where C is a purely positive clause
  2. $\mathcal{P}_{i+1}$ is obtained from $\mathcal{P}_i$ by means of an hyper extension step with an appropriate C.

- A **hyper tableaux refutation** is a derivation where $\mathcal{P}_n = \emptyset$

□

A hyper tableaux refutation of Example 2.4 is given in Figure 7 on page 41.

3 Usage

3.1 Syntax of the Input File

It follows the description of the syntax of input files of PROTEIN with extension `.tme` where the clause set to be proven is stated. Optionally theories can be included. The proof process of PROTEIN is controlled by flags stated in the input file.

3.1.1 Atoms (a)

An atom is a term with a functor beginning with a small letter, e.g. a, $tweety$, $bird(pinguin)$, $p(X)$ where arguments starting with small letters are treated as constants and those with a capital first letter as first order variables. Do not use atoms beginning with `not_`, `p_` or `protein_` or named like the predicates declared Prolog predicates by default (see Section 3.1.3) or one of the following:

> `any/0`, `query/0`, `restart/0`, `theory/3`, `begin/1`, `end/1`, `read/1`, `calculus/1`, `depth_increment/1`, `theory_costs/2`, `pttp_flag/2`, `prolog_pred/1`, `trace/1`, `,/2`, `;/2`, `:-/2`, `<-/2`, `->/2`, `|/2`, `#/2`

3.1.2 Literals (L)

A literal is an atom a or a negated atom written as $-a$, $-(a)$, $\sim a$, $\sim (a)$, or $true$, or $false$. All Prolog predicates are literals if they are declared to be `prolog_pred`, see Section 3.1.3.

3.1.3 Prolog Literals (L)

Some literals do not need to be proved with usual model elimination inference rules because they are either always true and used solely for side effects, such as `write/1`, `protein_answer/1`, or are Prolog builtins, or user-defined Prolog procedures the value of which can be determined by Prolog itself. You can tell PROTEIN not to generate code for proving those literals by declaring them Prolog predicates by stating

> `prolog_pred(`$Functor$`/`$Arity$`).`

It follows a list of predicates which are declared Prolog predicates by default.

> `!/0`, `;/2`, `==/2`, `>=/2`, `=</2`, `\+/1`, `\==/2`, `=../2`, `append/3`, `assert/1`, `call/1`, `delay/2`, `fail/0`, `false/0`, `flush/1`, `getval/2`, `incval/1`, `is/2`, `once/1`, `retract_all/1`, `setval/2`, `true/0`, `write/1`

3.1.4 Clauses (C)

From literals you can define clauses as follows:

syntax	meaning
L_1 `:-` $L_2, \ldots, L_n$.	$L_1 \vee \neg L_2 \vee \ldots \vee \neg L_n$
L_1 `<-` $L_2, \ldots, L_n$.	
$L_1; \ldots; L_m$ `:-` $L_{m+1}, \ldots, L_n$.	$L_1 \vee \ldots \vee L_m \vee \neg L_{m+1} \vee \ldots \vee \neg L_n$
$L_1, \ldots, L_m$ `:-` $L_{m+1}, \ldots, L_n$.	
$L_1; \ldots; L_m$ `<-` $L_{m+1}, \ldots, L_n$.	
$L_1, \ldots, L_m$ `<-` $L_{m+1}, \ldots, L_n$.	
$L_1; \ldots; L_n$.	$L_1 \vee \ldots \vee L_n$
$L_1, \ldots, L_n$.	
`?-` $L_1, \ldots, L_n$.	$\neg L_1 \vee \ldots \vee \neg L_n$

3.1.5 Theory Inference Rules (C)

Theory inference rules introduced in Definition 2.12 must be bracketed by

```
begin(theory).
      ⋮
end(theory).
```

They have the following syntax:

$L_1, \ldots, L_n$ `->` $L_{n+1}, \ldots, L_{n+m}$.
$L_1, \ldots, L_n$ `->` `false`.
$L_1, \ldots, L_n$ `->` $L_{n+1}, \ldots, L_{n+m}$ `|` *Guard*.
$L_1, \ldots, L_n$ `->` `false` `|` *Guard*.
where $n, m \geq 1$

Guard must be a predicate which can be executed directly and must be declared Prolog predicate (see 3.1.3). You can use directly executable predicates as a theory premise. If you want the theory conclusion to be a Prolog predicate use the following form:

$L_1, \ldots, L_n$ `-> false |` *Prolog predicate*.

3.1.6 Including Prolog-Code

If you want to define some Prolog predicates to be used for side effects in the proof process bracket them by

```
begin(prolog).
      ⋮
end(prolog).
```

These predicates are copied into the output file exactly as you wrote them. Predicates starting with `:-` are also executed when they are read. Do not use predicates starting with `p_`, `protein_`, or `h_`. Avoid the following global variables.

block, clausnum, ext, factor, factor_cut, inf, leafnr, more_depth, node, proof_depth, proof_time, red, red_cut, restart, theory, vertex, zarg

3.1.7 Flags

The scope of flags is always the whole input file, thus they can be set anywhere in the file. Some flags depend on a special calculus or another flag. PROTEIN gives you an error or a warning detecting inconsistencies. Some flags cause the setting of others. Undo this by following flag settings. A flag is set using

protein_flag(*Flag*,*Value*).

PROTEIN recognises flags and values of Table 1.

The use of most flags is described in Section 4, more detailed references are given in Table 1. Some flags have only technical effects which are described now.

- head_behind_prolog:
  If this flag is set to on, heads of input clauses which do not become head of a Prolog clause are sorted behind the first Prolog call if existing. This is useful if you want to call a Prolog predicate at the beginning of an extension step.

- mode:
  If this flag is set to pl0, the heuristically best flag setting for propositional logic is used which is:

  ```
  protein_flag(factorisation,cut).
  protein_flag(reduction,cut).
  protein_flag(regularity,nodelay).
  protein_flag(search,prolog).
  protein_flag(th_reduction,cut).
  protein_flag(th_regularity,nodelay).
  ```

  This flag overwrites flags set before. The values of the affected flags can be changed by following flag statements.

- sorting:
  If this flag is set to input, the order of the generated clauses corresponds to the order of clauses in the input file. Default value costs sorts the generated clauses with the same entry literal by costs. In Section 4.8 it is described what costs of clauses are, and how costs can be changed. The default costs of clauses are listed in Table 2 on page 42.

- translate:
  With the help of this flag you can determine whether all or only some Prolog clauses are generated. If this flag is set to value plain_theory the Prolog clauses for contrapositives (refer Section 4.1) are omitted.

flag	values	default	section	short description
ancestry_rme	on,off	off	4.3,4.4	ancestry RME
ancs	off,pos,neg,pos_dj, both_dj,both	both	4.2,4.4, 4.6	using of ancestor lists
answers	one,more,all	one	4.9	answer calculating
answer_set_handling	on,off	on	4.9	treating of identical answers
calculus	me,rme,hyper	me	4.1-4.4, 4.6,4.7	calculus
check_flags	*list of flag statements*	[1]		flag setting for consistency tests
definite_answers	on,off	off	4.9	calculating definite answers
depth_increment	*number*	1	4.8	increasing of the depth bound
factorisation	off,nocut,cut,both	cut	4.5	factorisation
head_behind_prolog	on,off	off	3.1.7	ordering body literals
mmr	off,wgcwa,gcwa	off		minimal model reasoning
mode	pl0,pl1	pl1	3.1.7	propositional or first order logic
query_reuse	on,off	on	4.1,4.3, 4.9	reusing the query
reduction	off,nocut,cut,both	both	4.2	reduction
regularity	off,nodelay,delay	delay	4.4	regularity
search	prolog,id_tree, id_term	id_tree	4.8	search strategy
selection_function	on,off	off	4.1	RME with selection function
sorting	input,costs	costs	3.1.7	sorting of the generated predicates
strict_rme	on,off	off	4.2,4.6	strict RME
th_nonewclauses	on,off	off	4.6	extending of a path to obtain premise literals
th_reduction	off,nocut,cut,both	both	4.6	theory reduction
th_regularity	off,nodelay,delay	delay	4.6	theory regularity
th_sidelit_anc	on,off	off	4.6	theory premise literals in ancestor lists
trace	off,info,ascii, xptree,tview, dynamic(*number*)	info	4.10	tracing
translate	all, plain_theory	all	3.1.7	translating mode

[1] [protein_flag(calculus,hyper)]

Table 1: flags and values

3.1.8 Other Statements

- The clauses and theory inference rules from Sections 3.1.4 and 3.1.5 can be assigned to weights so called costs explicitly by using the costs operator `#/2`:

  C `#` n.

  or

  C `#` n_1, n_2.

  where C is a clause or a theory inference rule, n, n_1, n_2 are the costs. The meaning of this is described in Section 4.8.

- `read(`*Name*`).`
  includes file *Name* into the input file. (*Name* must be the complete filename containing the extension.)

- Use `%` for line and `/* ··· */` for block comments.

3.1.9 Comment Standards

Semantic information about clauses may be given in a special syntax in comments which supports postprocessing of the proofs like generating a LaTeX output by the ILF server[5] [URL-ILF]. The following comment standards are optional.

- One complete clause per line.

- Non-theory clauses are enclosed in a

  ```
  % BEGIN_AXIOMS
        ⋮
  % END_AXIOMS
  ```

  bracket.

- A single non-theory clause is preceded by a comment which gives some semantic information in form of

  `% NAME:` *name*

  where *name* is a string enclosed by `" "`, a Prolog atom, or a Prolog term. E.g.

  `% NAME: "q is transitive"` or
  `% NAME: 'q is transitive'` or
  `% NAME: transitive(q)`

[5]The ILF server makes available some of the proof presentation tools of the interactive theorem prover ILF to users of other systems. ILF is developed at the Humboldt University Berlin. It's development is supported by the Deutsche Forschungsgemeinschaft.

- Theory inference rule are commented as follows:

  ```
  % NAME:  name
  % THEORY: clauselist
  C1
     ⋮
  Cn
  % NAME_END
  ```

  where *name* is a string enclosed by `" "`, a Prolog atom, or a Prolog term giving some semantic information, *clauselist* is the set of clauses of the horn theory for which the contrapositives C_1 to C_n are generated. *clauselist* has the syntax

  ```
  clauselist ::= '[' [ clause ] ']' |
                 '[' bclause { ',' bclause } ']'
  bclause    ::= '(' clause ')'
  clause     ::= lit [ ':-' lit ] |
                 lit ':- (' lit { ',' lit } ')'
  ```

  where `lit` is defined in Section 3.1.2.

  An sample comment for a theory:

  ```
  % NAME: transitive(p)
  % THEORY: [p(X, Z) :- (p(X, Y), p(Y, Z))]
  p(A, B), p(B, C) -> p(A, C).
  ~ p(A, C), p(B, C) -> ~ p(A, B).
  % NAME_END
  ```

Using the ILF-server strings occur in the LaTeX output as they are given, atoms and terms may be manipulated in the `TEXOP` section of the input mail like described in the ILF web pages at [URL-ILF]. Some sample files are given in Section 5.

3.2 Invoking

There are two types of invoking PROTEIN:

1. from within ECLiPSe
2. from the shell command line

1. After starting ECLiPSe the prover is loaded by typing

   ```
   [protein].
   ```

   Then you can either translate an input file and start the proving process at once using `protein(`*Name*`).`

(*Name* beeing the name of the input file without extension) or just translate the file using

`translate(`*Name*`).`

Once you have translated a file the proof can be run separately by first loading the generated Prolog file by typing

`[`*Name*`].`

and then starting the proof by

`proof.`

In order to create a stand-alone version of PROTEIN type

`make_protein_exec.`

2. First, you have to create the stand-alone program `protein` using `make_protein_exec/0` as described above. Then you can invoke

   `protein` *Name*

   in order to prove the file *Name*`.tme`.

3.3 Files

PROTEIN consists of the following files:

- `protein.pl`, the main module containing the predicates:
  - `translate/1` which compiles a given input file.
  - `protein/1` which compiles a given input file and starts the proof.
  - `make_protein_exec/0` which creates the stand-alone program `protein`.
- `protein2protop.pl`, the interface between PROTEIN and ProTop.
- `t_translate_1.pl`, the start of the compilation. The input file is processed. The clauses are read and transformed into an internal clause format. Code bracketed by `begin(prolog).` ⋯ `end(prolog).` is stored. All other informations of the input file like flags, declarations, etc. are made available for further use.
- `t_translate_2.pl`, in this second step of the translation the clauses are transformed depending on flag settings. Additional clauses are generated like query and restart clauses and added to the transformed input clauses. Also, all functors used in the input clauses are collected and stored for further use.

- t_translate_3.pl, now the clauses are transformed into an internal predicate structure, additional arguments like depth, ancestor, and aunt lists are added to the literals. All necessary contrapositives of the input clauses are generated.

- t_translate_4.pl, additional predicates e.g. test predicates for reduction steps are added. Also code for e.g. search strategies, tracing is added to each predicate.

- t_translate_5.pl, in this module the file with extension .pl is written. At the beginning several headers are copied to the file. In the end the predicates in the internal predicate structure are transformed into Prolog and added to the file. This file will then be executed to start the proving process. The headers are the following:

  – t_header.pl the main header presents the starting point of the proof, the predicate proof/0. It also contains code to control search strategies and to compute answers.

  – t_header_utils.pl, a collection of predicates called up during the proving process.

  – t_header_weight.pl, code to weight terms for iterative term deepening.

  – t_header_mmr.pl, code for minimal model reasoning.

  – t_header_info.pl, this header contains code for proof statistics.

  – t_header_trace.pl, predicates to write proof information.

  – t_header_tree.pl, predicates to build and translate the tree term.

  – t_header_dynamic.pl, predicates to display a proof tree during the proving process.

  – t_header_tview.pl, code to translate the tree term for tree viewer tview.

- t_utils.pl is a collection of useful utility predicates.

- t_definitions.pl, this file contains definitions of constants, input statements, flags, additional arguments, clause types, prolog literals, and messages.

- t_literal.pl, a collection of literal manipulating predicates.

- t_claus.pl which manages the internal clause structure.

- t_pred.pl which manages the internal predicate structure.

- README

4 Implementation

PROTEIN is designed according to *Stickel*'s PTTP-technique (**P**rolog **T**echnology **T**heorem **P**rover) introduced in [Sti88]. That means PROTEIN is based on the almost complete theorem prover Prolog. Changes are done to handle the non-horn case. By this technique the benefits of Prolog i.e. ECLiPSe are accessible to theorem proving because PROTEIN is implemented in ECRC's ECLiPSe Prolog.

PROTEIN compiles the clauses of the input file with extension `.tme` (stands for **t**heory **m**odel **e**limination) and translates them into Prolog code which is written into a file with the extension `.pl`. This file is loaded and executes the clauses according to the model elimination calculus or one of its variants described in Section 2.

% indexunification The unification procedure of Prolog must be replaced by unification with occur check. In ECLiPSe this is easily done by a flag.

In the next three subsections it is described how the inference rules extension, reduction, and restart are realized in PROTEIN.

4.1 Extension Steps

First, it is shown how the inference rule extension becomes reality in PROTEIN. Therefore all input clauses are transferred into Prolog clauses. Additionally contrapositives are needed, because in the model elimination calculus which is default of flag `calculus` every literal can equally well serve as an entry point into the clause (review Definition 2.5).

Example 4.1 (contrapositives) For clause $p(A,C) \leftarrow p(A,B), p(B,C)$ from Example 5.2 the following contrapositives are generated:

```
p(A,C,. . .,'_*') :-
   p(A,B,. . .,'_*'),
   p(B,C,. . .,'_*').

not_p(A,B,. . .,'_*') :-
   not_p(A,C,. . .,'_*'),
   p(B,C,. . .,'_*').

not_p(B,C,. . .,'_*') :-
   not_p(A,C,. . .,'_*'),
   p(A,B,. . .,'_*').
```

□

This example also shows how negation is treated, namely by making it part of the predicate name. Every literal gets additional arguments, number and usage depending on the flag setting. The constant argument `'_*'` just marks that these literals have those arguments. The other arguments are described in the following sections. All contrapositives of the input clauses in Example 2.4 are generated in Example 4.2.

Using calculus restart model elimination (flag `calculus` is set to `rme`) it follows immediately from the inference rule extension in Definition 2.7 that this calculus only needs those contrapositives of clauses which contain a positive literal in their heads. Only these contrapositives are generated. This is only one Prolog clause for the input clause in Example 4.1.

Case of calculus variant restart model elimination with selection function (flag `calculus` is set to `rme` and flag `selection_function` is switched `on`) the first positive literal of an input clause is selected and solely used as an entry literal (see Definition 2.10). Only one contrapositive of each clause is generated. The contrapositives for Example 2.4 for restart model elimination with selection function are listed in Example 4.3.

If there are queries[6] given in the input file, each is translated into a query clause[7]. For a model elimination calculus (value `me` of `calculus` flag) they are additionally picked up as purely negative clause. In this case the user is responsible for a sufficient set of queries for completeness. If no query is given in the input file, the clause set is transformed into a variant of goal normal form, described on page 5 by adding a query clause for each purely negative input clause.

If flag `query_reuse` is set `off` the query may be used only once in a proof. This is realized by generating only the query clause and no purely negative clause for each given query or purely negative input clause in the case of a normal form transformation. Note, that this setting makes all calculus variants incomplete except ancestry restart model elimination.

All proofs are started by calling `?- query(. . .,_*').`

4.2 Reduction Steps

For model elimination as second inference rule the reduction rule has to be implemented. Therefore you have to memorise subgoals proved so far in an additional argument of each Prolog clause. In PROTEIN exist two such ancestor lists for positive and negative literals. To implement the reduction step you have to check if the actual subgoal is member of an ancestor list. This is done by the predicate `p_unifiable/2` which works like `member/2`.

```
p_unifiable(El,[El|_]).
p_unifiable(El,[_,El|_]).
p_unifiable(El,[_,_,El|_]).
p_unifiable(El,[_,_,_|R]) :- p_unifiable(El,R).
```

If a reduction step does not affect variables it can be handled irrevocably with a cut. It is performed if the current literal is an identical complement of one of its

[6] an input clause of syntax `?-`$L_1, \ldots, L_m$, $m \geq 1$

[7] a clause of form $query \leftarrow B_1, \ldots, B_m$, $m \geq 1$

ancestors. For this test predicate `p_identical/2` is used which looks for identical members of a list. But this test alone is only complete in ground case. Otherwise both tests are used with priority to the cut version.

```
p_identical(El,[X|_])      :- El == X, !.
p_identical(El,[_,X|_])    :- El == X, !.
p_identical(El,[_,_,X|_]) :- El == X, !.
p_identical(El,[_,_,_|R]) :- p_identical(El,R), !.
```

In case of calculus strict restart model elimination (`calculus` flag is set to `rme`, flag `strict_rme` is switched `on`) reduction tests are only generated for positive entry literals, because reduction steps may only be applied at negative leafs (see Definition 2.8). Otherwise reduction tests are generated for every entry literal of the clause set. The last case is demonstrated in the following example:

Example 4.2 (Prolog clauses for model elimination) PROTEIN compiles the input clauses from Example 2.4 into the following Prolog clauses using the listed flags:

```
protein_flag(calculus,me).    %default
protein_flag(reduction,both). %default
protein_flag(ancs,both).      %default
protein_flag(factorisation,off).
protein_flag(regularity,off).
protein_flag(search,prolog).
protein_flag(trace,off).

p((_,NegAnc),'_*') :- p_identical(not_p,NegAnc), !.
p((_,NegAnc),'_*') :- p_unifiable(not_p,NegAnc).
p((PosAnc,NegAnc),'_*') :-
   not_q(([p|PosAnc],NegAnc),'_*').
p((PosAnc,NegAnc),'_*') :-
   q(([p|PosAnc],NegAnc),'_*').

q((_,NegAnc),'_*') :- p_identical(not_q,NegAnc), !.
q((_,NegAnc),'_*') :- p_unifiable(not_q,NegAnc).
q((PosAnc,NegAnc),'_*') :-
   not_p(([q|PosAnc],NegAnc),'_*').
q((PosAnc,NegAnc),'_*') :-
   p(([q|PosAnc],NegAnc),'_*').

not_q((PosAnc,_),'_*') :- p_identical(q,PosAnc), !.
not_q((PosAnc,_),'_*') :- p_unifiable(q,PosAnc).
not_q((PosAnc,NegAnc),'_*') :-
   not_p((PosAnc,[not_q|NegAnc]),'_*').
not_q((PosAnc,NegAnc),'_*') :-
   p((PosAnc,[not_q|NegAnc]),'_*').
```

```
not_p((PosAnc,_),'_*') :- p_identical(p,PosAnc), !.
not_p((PosAnc,_),'_*') :- p_unifiable(p,PosAnc).
not_p((PosAnc,NegAnc),'_*') :- not_q((PosAnc,[not_p|NegAnc]),'_*').
not_p((PosAnc,NegAnc),'_*') :- q((PosAnc,[not_p|NegAnc]),'_*').

query((PosAnc,NegAnc),'_*') :-
   p(([query|PosAnc],NegAnc),'_*'),
   q(([query|PosAnc],NegAnc),'_*').
```

□

The two clauses for reduction tests for each entry literal are followed by extension clauses of the same literal because Prolog requires clauses sorted by their heads. Besides the constant additional argument an argument for the ancestor lists `PosAnc` and `NegAnc` the lists for the already proven subgoals mentioned above is used. This example also shows how the ancestor lists are stored and extended.

You can manipulate the generation of the reduction tests by flag `reduction`. Values `off`, `nocut`, `cut`, and `both` enable you to have none, one, or both reduction tests. For propositional logic only the cut test is useful, otherwise both tests are used by default.

Flag `ancs` controls the usage of the ancestors which has five possible values: If the flag is switched `off` there is no additional argument for ancestor lists, and no reduction test is generated. For all other values an additional argument for the two ancestor lists is introduced. Case of value `pos` (`neg`) one list for positive (negative) ancestors is used, only for negative (positive) entry literals reduction tests are generated, and just for extension steps with positive (negative) entry literals the ancestors will be extended. If the flag is set to `both`, its default, both ancestor lists are used according to Example 4.2. The difference between `pos` and `pos_dj`, `both` and `both_dj` is that positive ancestors are only collected if they stem from disjunctive input clauses[8]. (Read [BauBrü96] for details). Example 4.14 illustrates the generation of clauses for value `neg` of flag `ancs`.

4.3 Restart Steps

For the calculus variants restart model elimination (flag `calculus` is set to `rme`) additionally the inference rule restart has to be implemented. This is done by calling `not_query` again (remember Definition 2.7). PROTEIN distinguishes between the starting point of a proof and restart steps by calling `not_restart` instead and having restart clauses[9] beside query clauses.

Every purely negative clause of the input file is translated into a restart clause. No additional purely negative clause is needed because it contains no literal for connections with negative leafs. (Extensions of positive leafs are forbidden following

[8]clauses with more than one positive literal

[9]clauses of form $restart \leftarrow B_1, \ldots, B_m$, $m \geq 1$

Definition 2.7). Additionally for every positive entry literal the invoking of a restart step is generated, a clause of form $A \leftarrow restart$.

Using the ancestry restart variant (flag `calculus` is set to `rme`, the value of flag `ancestry_rme` is `on`) restart steps are performed not only by calling `not_restart` but every negative ancestor literal (renember Definition 2.11). Therefore one more restart clause has to be generated which looks for candidates in the positive ancestor list[10]. To get shorter refutations the restart literal is searched from the leaf towards the top. Such a clause is shown in Example 4.3. If flag `ancestry_rme` is switched to its default `off` this clause is left out.

Example 4.3 (Prolog clauses for ancestry rme with selection function) PROTEIN translating the clauses from Example 2.4 with listed flag setting into following Prolog clauses:

```
protein_flag(calculus,rme).
protein_flag(ancestry_rme,on).
protein_flag(selection_function,on).
protein_flag(mode,pl0). %causes flag settings
protein_flag(factorisation,off).
protein_flag(regularity,off).
protein_flag(trace,off).

p((_,NegAnc),'_*') :- p_identical(not_p,NegAnc), !.
p((PosAnc,NegAnc),'_*') :-
   not_q(([p|PosAnc],NegAnc),'_*').
p((PosAnc,NegAnc),'_*') :-
   q(([p|PosAnc],NegAnc),'_*').

q((_,NegAnc),'_*') :- p_identical(not_q,NegAnc), !.
q((PosAnc,NegAnc),'_*') :-
   p(([q|PosAnc],NegAnc),'_*').

not_q((PosAnc,_),'_*') :- p_identical(q,PosAnc), !.
not_q((PosAnc,NegAnc),'_*') :-
   restart((PosAnc,[not_q|NegAnc]),'_*').

not_p((PosAnc,_),'_*') :- p_identical(p,PosAnc), !.
not_p((PosAnc,NegAnc),'_*') :-
   restart((PosAnc,[not_p|NegAnc]),'_*').

restart((PosAnc,NegAnc),'_*') :-
   member(Lit,PosAnc),
   Lit \== restart,
   Lit \== query,
   Lit =.. LitL,
```

[10]Negative leafs are stored in positive ancestor lists.

```
   append(LitL, [(PosAnc,NegAnc),'_*'], LongLitL),
   LongLit =.. LongLitL,
   call(LongLit).

restart((PosAnc,NegAnc),'_*') :-
   p(([query|PosAnc],NegAnc),'_*'),
   q(([query|PosAnc],NegAnc),'_*').

query((PosAnc,NegAnc),'_*') :-
   p(([query|PosAnc],NegAnc),'_*'),
   q(([query|PosAnc],NegAnc),'_*').
```

□

If you execute this Prolog code you get into an endless loop. You need the regularity restriction or a fair search strategy described in the following sections to avoid this. For entry literal `q` the second extension clause is missing (compare with Example 4.2). In clause $p, q \leftarrow$ the first positive literal is selected by the selection function and only one extension clause with the selected head literal is generated. The extension steps for negative entry literals are left out. Instead the reduction tests are followed by restart calls. Solely the restart calls are there for strict restart model elimination. Beside the query predicate two clauses of a restart predicate are generated. For non ancestry restart model elimination the first restart clause is deleted as shown in Example 4.4.

If flag `query_reuse` is set `off` the query may be used only one time, no restart may be carried out with literal `not_restart`. This is done by generating solely the query clause and no restart clause for each given query or purely negative input clause in the case of a normal form transformation. Only the ancestry restart clause is left if there. This makes the calculus incomplete beside using ancestry restart model elimination.

4.3.1 Delayed Restart Steps

Delayed restart model elimination is a variant of restart model elimination which enables PROTEIN to compute saturated tableaux which are model elimination trees with closed negative and open positive leafs. The aim of delaying is not to speed up the calculus. Saturated tableaux are usefull to compute the weak general closed world assumption and the general closed world assumption of a literal.

For this calculus variant all restart calls are delayed, the execution is interupted and will be continued later. In the meantime other open Goals are invoked. Delaying is implemented by an internal predicate of Eclipse called `delay/2`.

```
delay(Term,Goal)
```

delayes the goal `Goal` until a variable in `Term` is instantiated. Because of our aim to compute saturated tableaux, `Term` consists of one variable which is instantiated at the end of a query or restart clause. This delay variable is passed to all subgoals of the query or restart clause by an additional argument. Because of delaying of restart steps positive leafs are left open and negative leafs are tried to close. If it is possible to close all negative leafs the resulting path set contains only paths which end in positive leafs, a saturated tableau. Now, the delay variable is instantiated and all restart steps are woken up in an unknown order.

Example 4.4 (Prolog clauses for delayed restart model elimination)
Prolog clauses generated for Example 2.4 with flags from Example 4.3 except

```
protein_flag(ancestry_rme,off). %default
protein_flag(delayed_rme,on).

p((_,NegAnc),_,'_*') :- p_identical(not_p,NegAnc), !.
p((PosAnc,NegAnc),Delay,'_*') :-
   not_q(([p|PosAnc],NegAnc),Delay,'_*').
p((PosAnc,NegAnc),Delay,'_*') :-
   q(([p|PosAnc],NegAnc),Delay, '_*').

q((_,NegAnc),_,'_*') :- p_identical(not_q,NegAnc), !.
q((PosAnc,NegAnc),Delay,'_*') :-
   not_p(([q|PosAnc],NegAnc),Delay,'_*').
q((PosAnc,NegAnc),Delay,'_*') :-
   p(([q|PosAnc],NegAnc),Delay,'_*').

not_q((PosAnc,_),_,'_*') :- p_identical(q,PosAnc), !.
not_q((PosAnc,NegAnc),Delay,'_*') :-
       delay(Delay,restart((PosAnc,[not_q|NegAnc]),_,'_*')).

not_p((PosAnc,_),_,'_*') :- p_identical(p,PosAnc), !.
not_p((PosAnc,NegANc),Delay,'_*') :-
       delay(Delay,restart((PosAnc,[not_p|NegAnc]),_,'_*')).

query((PosAnc,NegAnc),_,'_*') :-
       p(([query|PosAnc],NegAnc),Delay,'_*'),
       q(([query|PosAnc],NegAnc),Delay,'_*'), !,
       p_unify(Delay,awake).

restart((PosAnc,NegAnc),_,'_*') :-
       p(([restart|PosAnc],NegAnc),_,'_*'),
       q(([restart|PosAnc],NegAnc),_,'_*'), !,
       p_unify(Delay,awake).
```

□

Predicate `p_unify/2` renames the unification predicate `=/2`. Reduction tests do not change exept the useless additional argument for the delay variable in the head literal. In extension steps the additional argument is added to head and body literals and thereby passed to subgoals. The execution of this Prolog code still loops due to lack of regularity or a fair search strategy.

As shown in Example 4.4 flag `delayed_rme` controlls delaying of restart steps by values `on` and default value `off`. All restart variants like ancestry restart model elimination, strict restart model elimination, restart model elimination with selection function may be combined with delaying restart steps. Even the hyper transformation may be modified by the delaying feature.

4.4 Regularity Steps

Next, two refinements of model elimination and its variants will be introduced which are the regularity restriction and factoring steps. Regularity is easy to implement and it is one of the more effective restrictions for model elimination and its variants. It means briefly that in the course of a proof it is never necessary to construct a tree where a literal occurs more than once along a path, the repetition of the same subgoal along a branch can be discarded. Unfortunately, the test has to be relaxed for restart model elimination. For instance, if a restart step is applied more than once for a path, the extension with the `not_restart` literal violates the regularity restriction. Thus, three different kinds of regularity will be defined according to [BauFurSto95].

Definition 4.5 (regularity)
Let p be a path written as follows (A^i positive B_k^j negative literals):
$p = \left\langle B_1^1, \ldots, B_{m_1}^1, A^1, B_1^2, \ldots, B_{m_2}^2, A^2, \ldots, A^{n-1}, B_1^n, \ldots, B_{m_n}^n \right\rangle$

- p is called **regular** iff

  1. $A^i \neq A^j$ for $1 \leq i, j, \leq n-1$, $i \neq j$

  2. $B_k^i \neq B_l^j$ for $1 \leq i, j \leq n$, $1 \leq k \leq m_i$, $1 \leq l \leq m_j$, $i \neq j \vee k \neq l$

- p is called **blockwise regular** iff **1.** and

  3. $B_k^i \neq B_l^i$ for $1 \leq i \leq n$, $1 \leq k, l \leq m_i$, $k \neq l$

- p is called **globally regular** iff **1.** and

  4. $B_k^i \neq B_l^j$ for $1 \leq i, j \leq n$, $1 \leq k \leq m_i$, $2 \leq l \leq m_j$, $i \neq j \vee k \neq l$

A path multiset is called regular (blockwise regular, globally regular) iff every path in it is regular (blockwise regular, globally regular). Similar a derivation is called regular (blockwise regular, globally regular) iff every path multiset is regular (blockwise regular, globally regular). □

Condition 1 states that all positive literals along a path are pairwise different. Condition 2 includes the analogous case for negative literals. But condition 3 says that negative literals inside blocks are pairwise different where a block means a smallest subpath delimited by positive literals or ends of the path. Condition 4 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.

The following combinations are known to be complete:

- plain model elimination and regularity
- ancestry restart model elimination and globally regularity
- all other variants of restart model elimination and blockwise regularity (proof in [BauFur94b])

There is no difference between regularity and globally regularity in the implementation in PROTEIN because regularity tests are not executed for literals following a restart. To implement the regularity steps you have to check whether the subgoal gets ever identical to one of its ancestors. Therefore Prolog clauses are generated which test whether the actual entry literal violates a regularity restriction. In case of blockwise regularity for positive entry literals ancestors lists are only examined until a `restart` literal is found.

For calculus model elimination (flag `calculus` set to `me`) or calculus ancestry restart model elimination (value `rme` of flag `calculus` and flag `ancestry_rme` switched `on`) predicate `p_never_identical/2` is used to implement the regularity test.

```
p_never_identical(_,[]).
p_never_identical(El,[X|R]) :-
   !, El ~= X,
   p_never_identical(El,R).
```

In the other restart model elimination calculus variants the blockwise predicate `p_never_identical_upto_restart/2` is used for regularity tests in Prolog clauses with positive[11] entry literals. For negative entry literals the predicate `p_never_identical/2` is used again.

```
p_never_identical_upto_restart(_,[]).
p_never_identical_upto_restart(_,[restart|_]) :- !.
p_never_identical_upto_restart(El,[X|R]) :-
   !, El ~= X,
   p_never_identical_upto_restart(El,R).
```

Both predicates use the difference operator `~=/2` to compare the literals which succeeds if the two literals cannot be unified. If they are unifiable but not identical the goal is delayed. This has the effect that the test even fails if the literals become identical by application of a substitution later on in the proof. The use of the two different test predicates is shown in the following example.

[11] The call to a positive head literal attempts to prove a negative leaf.

Example 4.6 (Prolog clauses for delayed regularity tests) Two different regularity tests generated for input clauses from Example 2.4 for a non ancestry but restart model elimination calculus with listed flags:

```
protein_flag(calculus,rme).
protein_flag(ancestry_rme,off).   %default
protein_flag(regularity,delay).   %default
protein_flag(ancs,both).          %default
protein_flag(factorisation,off).
protein_flag(search,prolog).
protein_flag(trace,off).

p(r,(PosAnc,NegAnc),'_*') :- !,
   p_never_identical_upto_restart(p,PosAnc),
   p(n,(PosAnc,NegAnc),'_*').

not_p(r,(PosAnc,NegAnc),'_*') :- !,
   p_never_identical(p,NegAnc),
   p(n,(PosAnc,NegAnc),'_*').
```

□

The first additional argument of the literals is used to store whether the test has already been called to control left recursion. To apply a regularity test this parameter must be unifiable with `r`. After the application the parameter is set to `n` to avoid recursion. Remember the restart clause for ancestry restart model elimination of Example 4.3. The first parameter of the literal which is used for the restart is instantiated with `n`. This avoids regularity tests for this literal and leads to globally regularity.

Delaying is too much overhead for the ground case. Therefore PROTEIN offers a not delayed version of the two regularity tests. For this predicate `p_identical/2` from Section 4.2 is used and a blockwise version of it named `p_identical_upto_restart/2`.

```
p_identical_upto_restart(_,[restart|_])     :- !, fail.
p_identical_upto_restart(El,[X|_])          :- El == X, !.
p_identical_upto_restart(_,[_,restart|_])   :- !, fail.
p_identical_upto_restart(El,[_,X|_])        :- El == X, !.
p_identical_upto_restart(_,[_,_,restart|_]) :- !, fail.
p_identical_upto_restart(El,[_,_,X|_])      :- El == X, !.
p_identical_upto_restart(El,[_,_,_|R]) :-
   p_identical_upto_restart(El,R).
```

If one of the last two predicates succeeds the regularity test fails. The use of the predicates is demonstrated in the following example:

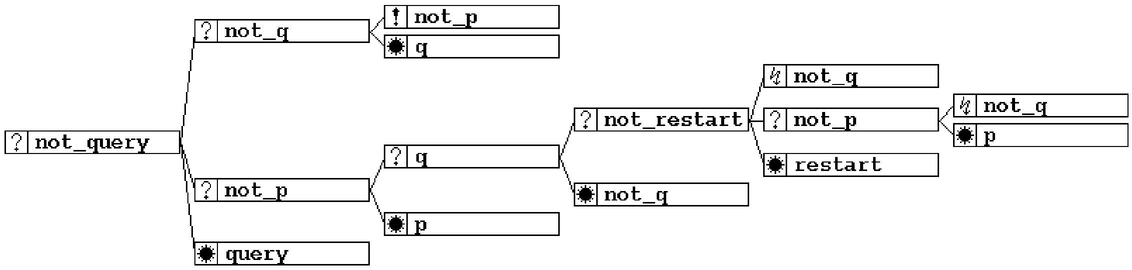


Figure 6: a rme refutation with selection function and factorisation for all-for-two

Example 4.7 (Prolog clauses for not delayed regularity tests) Two different regularity tests generated for input clauses from Example 2.4 for a non ancestry but restart model elimination calculus. The flag setting from Example 4.6 was used except `protein_flag(regularity,nodelay)`.

```
p(r,(PosAnc,_),'_*') :-
   !, p_identical_upto_restart(p,PosAnc), !, fail.
not_p(r,(_,NegAnc),'_*') :-  p_identical(p,NegAnc), !, fail.
```

□

Note, that for negative entry literals the predicate for delayed or not delayed tests never changes. Only for positive literals it depends on the calculus. For model elimination or ancestry restart model elimination the same test predicate is used for positive and negative entry literals.

As shown in Examples 4.6 and 4.7 flag `regularity` is used to switch between the delayed (value `delay`) and not delayed (value `nodelay`) version of the regularity test. Value `off` prevents from testing. In the last case you have no additional argument for the regularity test marker.

Analog to reduction steps flag `ancs` influences the generations of regularity tests. Value `off` stops the use of regularity tests. Values `pos` and `pos_dj` or `neg` where only one ancestor list is used causes regularity tests for negative or positive entry literals only. The values `both` and `both_dj` do not change the test generation.

4.5 Factoring Steps

The second optimisation of the calculi is factorisation. It permits to mark a leaf literal L_1 as solved if it can be unified with the literal of another node L_2 provided that the set of ancestors of L_2 is a subset of the ancestors of L_1. Additionally the unifier for L_1 and L_2 has to be applied to the path multiset. This is illustrated by Figure 6 which shows a restart model elimination refutation with selection function of Example 2.4 with factoring steps. The closing of a branch due to factorisation is marked with an exclamation mark. Factorisation avoids redundant proving of the same literal (compare with Figure 3).

To implement this test already proven subgoals have to be stored. In PROTEIN the subgoals are stored in two lists for positive and negative so called aunts stored in an additional argument of the generated clauses. The factorisation test is carried out by looking for unifiable members in an aunt list. Similar to reduction steps the test may be invoked with a cut if the compared literals are identical. For the two tests predicates `p_unifiable/2` and `p_identical/2` are used which were introduced in Section 4.2.

Example 4.8 (predicates for factorisation tests) Two different factorisation tests generated for input clauses from Example 2.4 with listed flags:

```
protein_flag(factorisation,both).
protein_flag(search,prolog).
protein_flag(trace,off).

p(n,_,(PosAunt,_),'_*') :- p_identical(p,PosAunt), !.
p(n,_,(PosAunt,_),'_*') :- p_unifiable(p,PosAunt).

query(n,(PosAnc,NegAnc),(PosAunt,NegAunt),'_*') :-
   p(r,([query|PosAnc],NegAnc),(PosAunt,NegAunt),'_*'),
   q(r,([query|PosAnc],NegAnc),([p|PosAunt],NegAunt),'_*').
```

□

The `query` clause illustrates how the proven subgoals are collected. This technique ensures that the subset property of the ancestors of the tested literals is fulfilled.

The tests are controlled by flag `factorisation`. You can determine the usage of none, one, or both tests by flag values `off`, `nocut`, `cut`, and `both`. If the value of the flag is `off` no additional argument form aunt lists is used. The default value of the flag is `cut`.

4.6 Theory Steps

Next, theory reasoning will be included in PROTEIN. This is realized by theory steps, if a theory is given in a `theory` environment of an input file.

Total theory inference rules are those with conclusio `false`. (The syntax is described in detail in Section 3.1.5.) They are handled similar to reduction steps because their application closes paths. Partial theory inference rules extend paths similar to extension steps.

For theory steps several kinds of Prolog clauses have to be generated. First of all, every given partial or total theory inference rule is transformed into several so called theory start clauses which are similar to the Prolog clauses for contrapositives for ordinary input clauses. Second, a predicate is defined that is invoked in the theory start clauses to prove the premise literals.

4.6.1 Theory Start Clauses

Every theory inference rule of form

$$L_1, \ldots, L_n \texttt{ -> } L_{n+1}, \ldots, L_{n+m} \,|\, Guard.$$

where the $L_1, \ldots, L_n$ are called premise literals and $L_{n+1}, \ldots, L_{n+m}$ conclusio literals, is transformed into n Prolog clauses of form

$$\begin{array}{ll} \overline{L_i} \leftarrow & \\ \quad theory(\underline{Flag}, [L_1, \underline{\ldots, L}_{i-1}, L_{i+1}, \ldots, L_n]\texttt{,\_)}, & \\ \quad Guard, \overline{L_{n+1}}, \ldots, \overline{L_{n+m}}. & 1 \leq i \leq n \end{array}$$

to start theory steps. The negated literals $\overline{L_i}$ and $\overline{L_{n+1}}, \ldots, \overline{L_{n+m}}$ and `theory/3` get additional arguments, the leftover premise literals $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$ do not. If $m = 1$ and $L_{n+1} =$ `false` the literal is left out. Similarly *Guard* is deleted when `true` or not given. The theory predicate `theory/3` contains a flag, the list of the rest of the premise literals, and as last argument the bracketed two ancestor lists which are used to save the ancestors at the end of the proving process of the premise literals. This is done if flag `th_sidelit_anc` is set to `on`. Following Definition 2.14 of theory model elimination the default of this flag is `off`. In the theory start clauses the `theory/3` call is used to prove the leftover premise literals. Afterwards a given *Guard* which is always a Prolog call is invoked. This *Guard* may be used for special unification procedures etc. If this call succeeds, a total theory step is finished, the path is closed. Otherwise, additionally the conclusio literals have to be proven.

In case of a variant of restart model elimination calculus (value `rme` of calculus flag) and flag `th_nonewclauses` set `on` the first argument of the `theory/3` predicate is instantiated with `n` for clauses with positive entry literals and total theory steps ($m = 1$ and $L_{n+1} =$ `false`). Otherwise the parameter is set to `y` which permits involving new input clauses to prove premise literals.

Flag `th_regularity` controls whether regularity test for every conclusio literal are added as first calls of the theory start clauses. For value `off` no test is added. Analogous to Section 4.4 predicates `p_never_identical/2` and `p_never_identical_upto_restart/2` are used for delayed regularity tests (value `delay` of `th_regularity`), and `p_identical/2` and `p_identical_upto_restart/2` for the not delayed regularity test (value `nodelay`) of the conclusio literals, where each second test predicate is used for positive literals and non ancestry but restart model elimination. Flag `regularity` has no effect on the regularity tests of conclusio literals, but flag `ancs` has. Value `off` avoids all regularity tests for conclusio literals. Values `pos` and `pos_dj` or `neg` where only one ancestor list is used causes regularity tests for negative or positive conclusio literals only. The values `both` and `both_dj` do not change the test generation. In Example 4.14 clauses for value `neg` of flag `ancs` are given.

Example 4.9 (theory start clauses) The listed input clauses and theory with given declarations

```
protein_flag(calculus,me).          %default
protein_flag(th_nonewclauses,off).  %default
protein_flag(ancs,both).            %default
protein_flag(th_sidelit_anc,off).   %default
protein_flag(search,prolog).
protein_flag(trace,off).

prolog_pred(nonar/1).

begin(theory).
p(X),-q(X) -> r(X).
-s(X) -> false | nonvar(X).
end(theory).
      ⋮
```

are translated into the following theory start clauses:

```
not_p(X,n,(PosAnc,NegAnc),Aunts,'_*') :-
   p_never_identical(not_r(X),NegAnc),
   theory(y,[not_q(X)],_,r,(PosAnc,[not_p|NegAnc]),Aunts,'_*'),
   not_r(X,r,(PosAnc,[not_p|NegAnc]),Aunts,'_*').

q(X,n,(PosAnc,NegAnc),Aunts,'_*') :-
   p_never_identical(not_r(X),NegAnc),
   theory(y,[p(X)],_,r,([q|PosAnc],NegAnc),Aunts,'_*'),
   not_r(X,r,([q|PosAnc],NegAnc),Aunts,'_*').

s(X,n,(PosAnc,NegAnc),Aunts,'_*') :-
   theory(y,[],_,r,([s|PosAnc],NegAnc),Aunts,'_*'),
   nonvar(X).
```

□

If flag `th_sidelit_anc` is set to `on` the 3rd parameter of the `theory/3` predicate, containing the ancestors from the end of the proving process of the premise literals, are used as ancestor lists for the conclusio literals in the body of the theory start clauses. If this flag is switched `off` the ancestor lists of the conclusio literals are not extended by premise literals or literals involved in the proving process of the premise literals.

If calculus strict restart model elimination is chosen (flag `calculus` set to `rme`, flag `strict_rme` to `on`) theory start clauses with negative entry literal are left out. For non strict restart model elimination predicates with negative heads are only allowed for total theory inference rules because of their similarity to reduction steps.

4.6.2 Theory Predicate

Next, the predicate `theory/3` will be defined. It proves recursively the rest of the premise literals given as second argument. The starting point are those kinds of clauses which are generated for both values of the first argument of the `theory/3` predicate. These are theory end clauses to finish the call of the theory predicate and theory reduction tests which prove premise literals analog to ordinary reduction tests.

The premise literals are proven if the second argument of the theory predicate is the empty list. Therefore the following clause is generated:

Example 4.10 (theory end clause)

```
theory(_,[],Ancs,. . .,Ancs,. . .,'_*').
```

□

Note, the ancestor lists in the `theory/3` predicate and in the additional arguments are unified. This saves the possibly extended ancestors for further use.

Second, theory reduction tests are introduced which are controlled by flag `th_reduction`. They work similar to the reduction tests introduced in Section 4.2 but for premise literals in the theory predicate. They use predicates `p_unifiable/2` for tests without and `p_identical/2` for tests with cut analog to reduction steps. The values `off`, `nocut`, `cut`, and `both` of flag `th_reduction` are used to generate none, one or both tests. Flag `reduction` has no effect on the test generation, but flag `ancs`. In case of value `off` no ancestor list is used and no reduction tests are generated. Values `pos`, `pos_dj` or `neg` where only one ancestor list is used causes reduction tests for negative or positive premise literals only. The other values of flag `ancs` have no effect on the theory reduction tests. In Example 4.14 clauses for value `neg` of flag `ancs` are listed.

Example 4.11 (theory reduction tests) The two different theory reduction tests for `theory/3` if both ancestor lists are used:

```
theory(Flag,[Lit|PremR],ThAncs,
      . . .,(PosAnc,NegAnc),. . .,'_*') :-
   p_neg_shortlit(Lit,PosNeg,NegLit),
   (PosNeg == pos ->
      p_identical(NegLit,NegAnc)
   ;
      p_identical(PosLit,PosAnc)), !,
   theory(Flag,PremR,ThAncs,
         . . .,(PosAnc,NegAnc),. . .,'_*').

theory(Flag,[Lit|PremR],ThAncs,
      . . .,(PosAnc,NegAnc),. . .,'_*') :-
   p_neg_shortlit(Lit,PosNeg,NegLit),
   (PosNeg == pos ->
      p_unifiable(NegLit,NegAnc)
```

```
;
   p_unifiable(PosLit,PosAnc)),
theory(Flag,PremR,ThAncs,
       . . .,(PosAnc,NegAnc),. . .,'_*').
```

□

In the next two kinds of clauses of predicate `theory/3` the first parameter ist always instantiated with `y` because they involve new clauses in the proving process of the premise literals. The path is extended to obtain premise literals. This are theory Prolog clauses which are used to prove literals which shall be proven by Prolog itself, and theory extension clauses which prove premise literals by extending them with other input clauses.

The next clause of the theory predicates is to call Prolog literals, literals declared Prolog (confirm Section 3.1.3). They need no special treatment and are just called. Those literals are enclosed when read, so they are easy to detect during the proof process.

Example 4.12 (theory Prolog clauses)

```
theory(y,[protein_prolog(Call)|PremR],ThAncs,. . .,'_*') :-
   Call,
   theory(y,PremR,ThAncs,. . .,'_*').
```

□

Finally, the theory extension clauses of the `theory/3` predicate will be described. Therefore every contrapositive of the input clauses is translated into a clause to extend a premise literal. In detail a contrapositive of form

L_1 `:-` $L_2, \ldots, L_n$.

is translated into a clause

`theory(y,[`L_1`|PremR],ThAncs) :- theory(y,PremR,ThAncs),`$L_2, \ldots, L_n$.

where `theory/3` and the $L_2, \ldots, L_n$ get additional arguments.

Example 4.13 (theory extension clauses) The contrapositives of example 4.1 are translated into the following Prolog clauses using default values of all flags except:

```
protein_flag(search,prolog).
protein_flag(trace,off).

theory(y,[p(A,C)|PremR],ThAncs,_,Ancs,(PosAunt,NegAunt),'_*') :-
   theory(y,PremR,ThAncs,_,Ancs,(PosAunt,NegAunt),'_*'),
   p(A,B,_,Ancs,(PosAunt,NegAunt),'_*'),
   p(B,C,_,Ancs,([p(A,B)|PosAunt],NegAunt),'_*').
```

```
theory(y,[not_p(A,B)|PremR],ThAncs,_,Ancs,(PosAunt,NegAunt),'_*') :-
   theory(y,PremR,ThAncs,_,Ancs,(PosAunt,NegAunt),'_*'),
   not_p(A,C,_,Ancs,(PosAunt,NegAunt),'_*'),
   p(B,C,_,Ancs,(PosAunt,[not_p(A,C)|NegAunt]),'_*').

theory(y,[not_p(B,C)|PremR],ThAncs,_,Ancs,(PosAunt,NegAunt),'_*') :-
   theory(y,PremR,ThAncs,_,Ancs,(PosAunt,NegAunt),'_*'),
   not_p(A,C,_,Ancs,(PosAunt,NegAunt),'_*'),
   p(B,C,_,Ancs,(PosAunt,[not_p(A,C)|NegAunt]),'_*').
```

□

4.7 Hyper Tableaux Transformation

To proof the input clauses according to the hyper tableaux calculus described in Section 2.4 the input clauses are translated into theory inference rules in such a manner, that theory restart model elimination behaves like a hyper tableaux calculus.

Setting flag `calculus` to value `hyper` causes, first, the setting of several flags to choose the right calculus variant, and second, a transformation of the input clauses. The affected flags are:

```
protein_flag(calculus,rme).
protein_flag(strict_rme,on).
protein_flag(reduction,nocut).
protein_flag(regularity,off).
protein_flag(ancs,neg).
protein_flag(th_nonewclauses,on).
```

Next, the input clause transformation is described: theory inference rules are not changed, from query clauses the `query` literal is deleted. Then a clause

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

is transferred into a theory inference rule

$$\texttt{not\_any}, P_1, \ldots, P_k, B_1, \ldots, B_{m'} \rightarrow A_1, \ldots, A_{n'}$$

where `not_any` is a new predicate, and P_i are literals declared Prolog. `not_any` is introduced to circumvent the link condition of model elimination and its variants. By default, the costs of the theory inference rule is n' the number of positive literals which are not declared Prolog and 0 which can be changed as described in Section 4.8. Finally a new query clause is added.

$$\texttt{query} \leftarrow \texttt{any}$$

Additionally the default costs of the following clause types are changed:

```
c_set_default_costs(query,0).
c_set_default_costs(restart,0).
c_set_default_costs(th_start,1,0).
```

Example 4.14 (hyper tableaux transformation) The hyper tableaux transformation for clauses from example 2.4:

```
not_any(_,(_,NegAunt),'_*') :- p_identical(not_any,NegAunt), !.
not_any((PosAnc,NegAnc),Aunts,'_*') :-
   restart((PosAnc,[not_any|NegAnc]),Aunts,'_*').

p(_,(PosAunt,_),'_*') :- p_identical(p,PosAunt), !.
p((_,NegAnc),_,'_*') :- p_unifiable(not_p,NegAnc).

q(_,(PosAunt,_),'_*') :- p_identical(q,PosAunt), !.
q((_,NegAnc),_,'_*') :- p_unifiable(not_q,NegAnc).

any(_,(PosAunt,_),'_*') :- p_identical(any,PosAunt), !.
any((_,NegAnc),_,'_*') :- p_unifiable(not_any,NegAnc).
any(Ancs,Aunts,'_*') :- theory(y,[p,q],_,Ancs,Aunts,'_*').
any((_,NegAnc),Aunts,'_*') :-
   \+ p_identical(not_p,NegAnc),
   theory(y,[q],_,(_,NegAnc),Aunts,'_*'),
   not_p((_,NegAnc),Aunts,'_*').
any((_,NegAnc),Aunts,'_*') :-
   \+ p_identical(not_q,NegAnc),
   theory(y,[p],_,(_,NegAnc),Aunts,'_*'),
   not_q((_,NegAnc),Aunts,'_*').
any((_,NegAnc),(PosAunt,NegAunt),'_*') :-
   \+ p_identical(not_p,NegAnc),
   \+ p_identical(not_q,NegAnc),
   theory(y,[],_,(_,NegAnc),(PosAunt,NegAunt),'_*'),
   not_p((_,NegAnc),(PosAunt,NegAunt),'_*'),
   not_q((_,NegAnc),(PosAunt,[not_p|NegAunt]),'_*').

not_p(_,(_,NegAunt),'_*') :- p_identical(not_p,NegAunt), !.
not_p((_,NegAnc),Aunts,'_*') :- restart((_,[not_p|NegAnc]),Aunts,'_*').

not_q(_,(_,NegAunt),'_*') :- p_identical(not_q,NegAunt), !.
not_q((_,NegAnc),Aunts,'_*') :- restart((_,[not_q|NegAnc]),Aunts,'_*').

theory(y,[protein_prolog(Call)|PremR],_,(_,NegAnc),Aunts,'_*') :-
   Call,
   theory(y,PremR,_,(_,NegAnc),Aunts,'_*').
theory(Flag,[Lit|PremR],_,(_,NegAnc),Aunts,'_*') :-
   p_neg_shortlit(Lit,pos,NegLit),
   p_identical(NegLit,NegAnc), !,
```

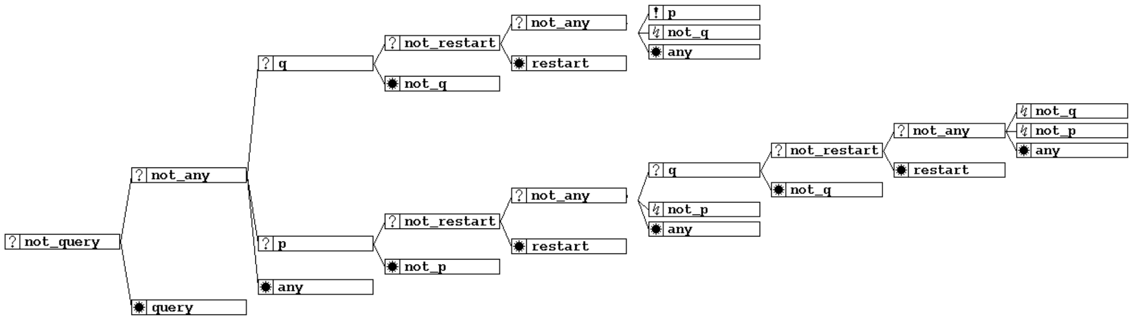


Figure 7: a hyper tableaux refutation for all-for-two

```
   theory(Flag,PremR,_,(_,NegAnc),Aunts,'_*').
theory(_,[]_,_,_,'_*').

query(Ancs,Aunts,'_*') :- any(Ancs,Aunts,'_*').
restart(Ancs,Aunts,'_*') :- any(Ancs,Aunts,'_*').
```

□

A derivation tree of the transformed clauses is given in Figure 7.

4.8 Search Strategies

In the previous steps of using PROTEIN we used Prolog's depth first search strategy by stating `protein_flag(search,prolog).` But this search strategy is unfair. A better choice is breadth first search, but this is too expensive to implement in Prolog. A compromise is tree depth bounded iterative deepening where you do depth first search until you reach a depth bound. This bound is increased step by step by a depth increment. Every step of the proof gets some costs which are subtracted from a depth potential of each depth bound. If there is not enough depth potential left backtracking will be invoked.

Costs of proof steps consist of two parameters which are numbers. To execute a proof step the depth resources must be greater or equal the sum of the two parameters. Afterwards the first value is subtracted from the depth resources. In Table 2 you see that usually the second number is 0.

Use Flag `search` to switch between Prolog's depth first search (value `prolog`) and iterative tree deepening (default value `id_tree`). To change the size of the steps increasing the depth bound use flag `depth_increment` whose value is a number (default `1`). The proof starts with this depth bound and increases the depth bound by that number.

There are two ways of changing the costs of a single input clause or theory inference rule using the `#`-operator.

C `#` n.

proof step	costs	Type(s)
extension step with facts	0,0	fact
extension step with other clauses	1,0	ext, query
reduction step	0,0	red, red_cut
regularity test	0,0	reg
factorisation test	0,0	factor, factor_cut
restart step	1,0	restart
theory reduction test	0,0	th_red, th_red_cut
theory extension step with facts	0,0	th_fact
theory extension step with other clauses	1,1	th_start
extension steps to prove premise literals	2,0	th_ext

Table 2: costs of proof steps

changes the first number of the costs to n, the second is still to its default. Similar

C `#` n_1, n_2.

changes both numbers of the costs of C. Changing the default costs of proof steps is more complicated. Therefore an internal predicate of PROTEIN is used.

Example 4.15 (changing the default costs of proof steps) The following example shows how to change one or both parameters of a proof step with type t. For types consult Table 2.

```
begin(prolog).
:-(protein_mode(compile) -> c_set_default_costs(t,n); true).
:-(protein_mode(compile) -> c_set_default_costs(t,n1,n2); true).
    ⋮
end(prolog).
```

□

The search strategy is compiled into the generated predicates by introducing another additional arguments being used as current depth. The test for enough depth resources is added to each predicate. Failing of the test causes backtracking for "cheaper" steps. In this case the global variable `more_depth` is set. If this is never done during a proof attempt for a given depth bound no proof is possible, PROTEIN stops. If `more_depth` is set and no proof with the actual depth bound was found the bound is increased and another proof attempt is started.

Example 4.16 (clauses for iterative tree deepening) The first entry literal of the predicates from Example 4.2 with default value of flag `search` and `depth_increment`. Additionally the setting from Example 4.2 is used.

```
p(_,(_,NegAnc),'_*') :- p_identical(not_p,NegAnc), !.
p(_,(_,NegAnc),'_*') :- p_unifiable(not_p,NegAnc).
```

```
p(Depth1,(PosAnc,NegAnc),'_*') :-
   (Depth1 >= 1 ->
      Depth2 = Depth1 - 1
   ;
      setval( more_depth, 1), !, fail),
   not_q(Depth2,([p|PosAnc],NegAnc),'_*').
p(Depth1,(PosAnc,NegAnc),'_*') :-
   (Depth1 >= 1 ->
      Depth2 = Depth1 - 1
   ;
      setval( more_depth, 1), !, fail),
   q(Depth2,([p|PosAnc],NegAnc),'_*').
```

□

For calculus hyper tableaux (`calculus` flag set to `hyper`) a further search strategy is offered, term bounded iterative deepening, which increases iteratively the term depth of literals. This strategy is used if flag `search` is set to `id_term`. The term depth of the conclusio literals of a theory start clause are weighted, and the result must be less or equal the actual depth bound. The weight of literal lists is calculated by predicate `p_weight/2`.

```
p_weight([],0).
p_weight([Lit|LitR],Weight) :-
   h_weight_term(Lit,TermWeight),
   p_weight(LitR,ZwiWeight),
   max(TermWeight,ZwiWeight,Weight), !.

h_weight_term(Lit,0) :- atomic(Lit), !.
h_weight_term(Lit,0) :- var(Lit), !.
h_weight_term(Lit,Depth) :-
   functor(Lit,_,Arity),
   h_weight_funct(Lit,1,Arity,Arity,Depth).

h_weight_funct(_,I,Max,Depth,Depth) :- I > Max, !.
h_weight_funct(Lit,I,Max,InDepth,OutDepth) :-
   arg(I,Lit,Arg),
   h_weight_term(Arg,ArgDepth),
   ZwiDepth is InDepth + ArgDepth,
   I1 is I + 1,
   h_weight_funct(Lit,I1,Max,ZwiDepth,OutDepth).
```

A further Prolog clause with entry literal `any` has to be generated now to increase the depth bound, if no proof could be found for the actual one. Similar to iterative tree deepening the value of the flag `depth_increment` determines the start bound and the size of the depth incrementation steps. This clause is moved at the end of the Prolog clauses with entry literal `any`.

Example 4.17 (clauses for iterative term deepening) For Example 2.4 the following Prolog clauses with entry literal any are generated for the listed flags:

```
protein_flag(mode,pl0).
protein_flag(calculus,hyper).
protein_flag(search,id_term).
protein_flag(depth_increment,1). %default
protein_flag(trace,off).

any(n,(_,NegAnc),_,'_*') :- p_identical(not_any,NegAnc), !.
any(n,(_,_,_,NegAnc,_,_,_,_,'_*') :-
   p_unifiable(not_any,NegAnc).
any(_,_,_,NegAnc,_,_,_,_,'_*') :-
   theory(y,[p,q],_,_,_,_,_,NegAnc,_,_,_,_, '_*').
any(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*') :-
   p_never_identical_upto_restart(not_p,NegAnc),
   theory(y,[q],_,_,_,_,_,NegAnc,_,_,_,_,'_*'),
   p_weight([not_p],Weight),
   Depth >= Weight,
   not_p(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*').
any(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*') :-
   p_never_identical_upto_restart(not_q,NegAnc),
   theory(y,[p],_,_,_,_,_,NegAnc,_,_,_,_,'_*'),
   p_weight([not_q],Weight),
   Depth >= Weight,
   not_q(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*').
any(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*') :-
   p_never_identical_upto_restart(not_p,NegAnc),
   p_never_identical_upto_restart(not_q,NegAnc),
   theory(y,[],_,_,_,_,_,NegAnc,PosAunt,NegAunt,_,_,'_*'),
   p_weight([not_p,not_q],Weight),
   Depth >= Weight,
   not_p(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*'),
   not_q(_,Depth,_,NegAnc,PosAunt,[not_p|NegAunt],_,_,'_*').
any(_,Depth,_,NegAnc,PosAunt,NegAunt,_,_,'_*') :-
   Depth1 is Depth + 1,
   write(Depth1),
   write(', '),
   flush(user),
   any(_,Depth1,_,NegAnc,PosAunt,NegAunt,_,_,'_*').
```

□

4.9 Answers

Theorem provers usually do not supply answers besides "yes" or "no". They can semi-decide whether a given clause set is unsatisfiable. Theorem provers using model

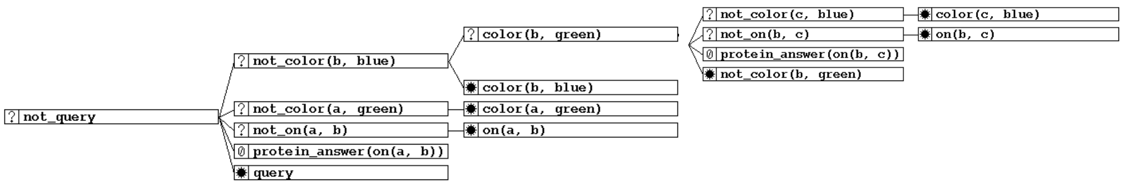


Figure 8: a model elimination refutation with answers for blocks world

elimination can be used as answer complete interpreters for disjunctive logic programming. For details and proofs for answer completeness of model elimination and the restart variants consult [BauFurSto95].

At the beginning I want to give a formal definition of the subject of interest:

Definition 4.18 (answer)
For a given clause set M and query clause $\leftarrow Q$ where Q stands for the disjunction of negative literals and $\sigma_1, \ldots, \sigma_m$ substitutions for the variables of Q, $Q\sigma_1 \vee \ldots \vee Q\sigma_m$ is an answer for M. □

PROTEIN offers a predicate `protein_answer/1` to collect substitutions of variables. If you use a query like

```
?- protein_answer((X,Y)),p(X,f(Y),Z).
```

PROTEIN displays you the instantiations of the variables `X` and `Y` at the end of a refutation. The predicate may be used in any input clause or premise or guard of a theory inference rule, but usually it is used in the query. (There is no need to negate this predicate in the input file.) For optimising it is useful to set the answer predicate as first literal of the clause. The definition of the predicate uses corouting techniques as follows:

```
protein_answer(Term) :- delay(true,protein_answer(Term)).
```

At the end of the proof all delayed answers are collected and printed. If flag `answer_set_handling` is set to its default value `on` duplicates are removed from the collected answers, otherwise (value `off`) they are not. See Example 4.19 for an example output of answers. Figure 8 shows a refutation tree with answer predicate for Example 5.4.

Occasionally one is interested in the question whether a given clause set admits a definite answer, i.e. an answer which is a single conjunction of atoms, but no disjunction. 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. This is done by flag `query_reuse` switched `off` whose effects to the generated clauses are described in Sections 4.1 and 4.3. Then definite answers are obtained. However, such a restriction is usually incomplete. For details see [BauFurSto95]. Using ancestry model elimination completeness is recovered. An other method for computing definite answers is to unify all occurrences of answer literals with each other during the proof process using flag `definite_answers` set `on`. This can be used in every variant of model elimination not only in the ancestry restart variant.

It follows the definition of the answer predicate for flag `definite_answers` set to `on`:

```
protein_answer(Term) :-
   delayed_goals(DL),
   \+ member( protein_answer(_), DL)),
   delay(true,protein_answer(Term)).
protein_answer(Term) :-
   delayed_goals(DL),
   member(protein_answer(Term),DL)).
```

The flags to compute definite answers are useful for Example 5.5. They avoid the computation of trivial answers like

```
knave / knave ; knave / knight ; knight / knave ; knight / knight
```

and force the computation of the solution of the puzzle.

```
knave / knight
```

Using the answer predicate, you might want not only the first answer found, but all possible answers. This is done by invoking backtracking at the end of a proof until another refutation is found. With help of flag `answers` you determine how many answers will be computed. The default value of this flag is `one` which prompts one answer and stops. Flag `answer` set to value `more` handles answers analog to Prolog. At the end of a proof one more answer is computed after typing ;. Value `all` lists all computed answers continuously.

Example 4.19 (answers for blocks world) It follows the output of PROTEIN for Example 5.4 which uses flag `protein_flag(answers,all)`:

```
********************    PROTEIN V1.18   ********************

translating bw.....
bw.pl      compiled traceable 25508 bytes in 0.18 seconds

proving bw...
tree depth 1, 2, 3, +++ proof found +++

Answers
-------
on(a, b) or
on(b, c)

Level +++ proof found +++
```

```
Answers
-------
on(b, c) or
on(d, b)

Level +++ proof found +++

Answers
-------
on(a, b) or
on(b, c)

Level +++ proof found +++

Answers
-------
on(b, c) or
on(d, b)

Level --- proof not possible ---
--- failure ---

bye
```

□

4.10 Tracing

Sometimes you are not solely interested whether it is possible to find a proof or not or in computed answers. You want statistics and a graphic representation of the derivation tree. PROTEIN offers several possibilities to retrieve such informations. Therefore flag `trace` is used to choose a special kind of tracing. Of cause collecting information during a proof slows it down.

If flag `trace` is switched `off` no information but `proof found`, `proof not possible`, or `failure` and optionally computed answers are printed on screen. The output `proof not possible` is given if no new depth resources have been demanded during a proof of a fixed depth bound in case of iterative tree deepening. For details review Section 4.8.

Choosing value `info` of flag `trace` makes PROTEIN counting proof steps and timing the proof process. Additionally the depth of the derivation is stored in case of an iterative deepening search strategy. First, the steps used solely for the derivation are computed. Second, all steps ever tried are counted, even if they are backtracked during the proof process. This is the so called number of inferences. After the proof process the derived data is printed on screen.

Example 4.20 (proof statistics) An output for Example 5.2:

```
Proof Statistics
----------------
Proof Time:  4.150000 sec
Proof Depth: 6
Extension Steps : 11
Restart Steps:    3
Reduction Steps:  5
Factoring Steps:  1
Inferences:       23927
```

□

This is implemented by adding code for every Prolog clause to increment global variables of ECLiPSe. The incrementation of the variables for the kinds of used proof steps for a refutation, concerning the variables `ext` for extension steps, `restart` for restart steps, `red` for reduction steps without cut, `red_cut` for reduction steps with cut, `factor` for factoring steps without cut, `factor_cut` for factoring steps with cut, and `theory` for theory steps, must be undone if backtracking is invoked. Therefore predicate `p_count/1` is implemented which follows. The global variable `inf` is used to count the inferences. In Example 4.21 Prolog clauses are given containing code for statistics.

```
p_count(Type) :- incval(Type).
p_count(Type) :- decval(Type), !, fail.
```

If another value of flag `trace` is given informations about the proof process are retrieved like if flag value `info` is chosen. Additionally a proof tree is built during the proving process. There are two kinds of generating proof trees.

One is doing it dynamically during the proving process. This is done if value `dynamic(`n`)` of flag `trace` is chosen, where n is a positive integer or float to indicate the seconds of slowing down the prooving process after displaying one node of the tree on screen. The proof tree is displayed with tree viewer `tview` which allows the communication with PROTEIN with the help of pipes. A call to predicate `p_dynamic/9` is added to each generated Prolog predicate (see Example 4.21). This predicate generates messages which are piped to the tree viewer which displays or deletes the nodes according to them. One more additional argument is introduced to give information to subgoals. With help of this kind of tracing you have a good image about how PROTEIN behaves, but it is very slow.

For all other values of flag `trace` one more additional argument of each literal is used to store a tree term which is recursively generated. A call to predicate `p_trace/6` is added to each Prolog clause to build the new tree term of this clause from the tree terms in the body literals.

Example 4.21 (clauses containing tracing code) The following clauses illustrate the generation of additional code to retrieve statistic data and generate a proof tree. Either predicate `p_dynamic/9` or `p_trace/6` is used.

```
% reduction step with cut
p(. . .,(_,NegAnc),. . .,Trace,'_*') :-
   p_identical(not_p,NegAnc), !,
%  p_dynamic(y,'reduction with cut',
%                 3,violetred,_,not_p,Trace,_,1),
   incval(inf),
   p_count(red_cut),
   p_trace(red_cut,0,0,p,[],Trace).

% factorisation test with cut
p(. . .,(PosAunt,_),. . .,Trace,'_*') :-
   p_identical(p,PosAunt), !,
%  p_dynamic(y,'factorisation with cut',
%                 5,indianred,_,not_p,Trace,_,1),
   incval(inf),
   p_count(factor_cut),
   p_trace(factor_cut,0,0,p,[],Trace).

% extension step
p(. . .,Trace,'_*') :-
   incval(inf),
%  p_dynamic(n,extension,
%                 1,steelblue,p,not_p,Trace,SubTrace,1),
   q(. . .,SubTrace,'_*'),
   p_count(ext).
   p_trace(ext,2,1,p,[SubTrace],Trace).

% theory step
p(. . .,(PosAnc,NegAnc),. . .,Trace,'_*') :-
   incval(inf),
%  p_dynamic(n,'partial theory inference',
%                 2,gold4,a,not_a,Trace,SubTrace,1),
%  Trace = SubTrace1, SubTrace = SubTrace2,
   p_never_identical(not_r(X),NegAnc),
   theory(y,[q],. . .,([p|PosAnc],NegAnc),. . .,SubTrace1,'_*'),
   not_r(. . .,([q|PosAnc],NegAnc),. . .,SubTrace2,'_*'),
   p_count(theory),
   p_trace(th_fact,1,2,not_b,[SubTrace1,SubTrace2],Trace).
```

□

After the proof process the tree term generated by predicate `p_trace\6` is translated into a format for a special tree viewer. Actually, three possibilities are implemented.

- `internal`:
  The tree term is printed as constructed in the additional argument with a final point in a file with extension `.tree`.

Example 4.22 (output of the tree term) Tree term of Example 2.4.

```
trc(query, info(8, query, 4, 1), [
   trc(p, info(5, ext, 1, 1), [
      trc(not_q, info(5, ext, 2, 2), [
         trc(not_p, info(5, red_cut, 0, 0), [])])]),
   trc(q, info(8, ext, 1, 2), [
      trc(not_p, info(8, ext, 3, 1), [
         trc(not_q, info(8, red_cut, 0, 0), [])])])]).
```

□

Syntax of this output file:

```
prooftreeterm ::= treeterm '.'
treelist      ::= '[' [ treeterm { ',' treeterm }] ']'
treeterm      ::= 'trc(' lit ',' infoterm ',' treelist ')'
infoterm      ::= 'info(' num ',' type ',' num ',' num ')'
lit           ::= atom | 'not_'atom
```

where `atom` is defined in Section 3.1.1, `num` is a positive number, and `type` is given in table 2 on page 42. The first number in the `infoterm` denotes the number of the inference, the second the clause number, the third the literal number of that clause. Counting always starts at 1 instead of 0. The last two numbers are not always used and are set to `0` by default.

- `ascii`:
  The tree term is printed in a more readable form in a file with extension `.tree`.

Example 4.23 (output of an ascii tree) Tree term of Example 2.4.

```
not_query  info(6, query, 4, 1)
not_p  info(3, ext, 1, 1)
   q  info(3, ext, 2, 2)
      p  info(3, red_cut, 0, 0)
not_q  info(6, ext, 1, 2)
   p  info(6, ext, 3, 1)
      q  info(6, red_cut, 0, 0)
```

□

- `xptree`:
  The tree term is translated into a syntax for the Munich `xptree` module. The translated term is written into a file with extension `.tree`. Afterwards `xptree` is called with that file. A picture of the `xptree` viewer for example 2.4 is given in Figure 9. The tree presentation is very compact, but literals with many arguments are shorten.

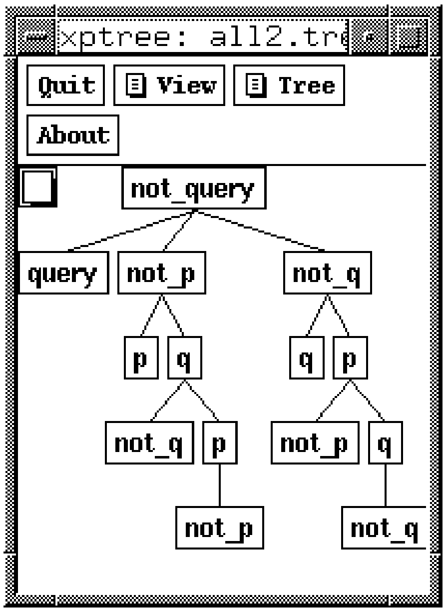


Figure 9: refutation tree presented by tree viewer xptree

- `tview`:
  The term is translated for the tree viewer `tview` from the Berlin deduction group. Therefore a file with extension `.tview` is written and invoked with `tview`. For comparison both windows presented by this tree viewer are shown in Figure 10. All graphical representations of derivation trees of the previous sections are generated with `tview`. Many options are available for `tview`. The disadvantage of this tree viewer is that all literal boxes have the same size.

Additionally for all three flag values the input clauses, flag setting, proof statistics, answers etc are written in a file with extension `.trc` which is shown on screen by the Unix demand `less`.

Example 4.24 (trace file for all-for-two)

```
********************    PROTEIN V2.05   ********************

--------------------    input clauses   --------------------

"Clause " No. 1 : [p, q]
"Clause " No. 2 : [p, not_q]
"Clause " No. 3 : [not_p, q]
"Query  " No. 4 : [query, not_p, not_q]
"Clause " No. 4 : [not_p, not_q]

--------------------        flags       --------------------

protein_flag(ancestry_rme, off).
   . . .
protein_flag(translate, all).
```

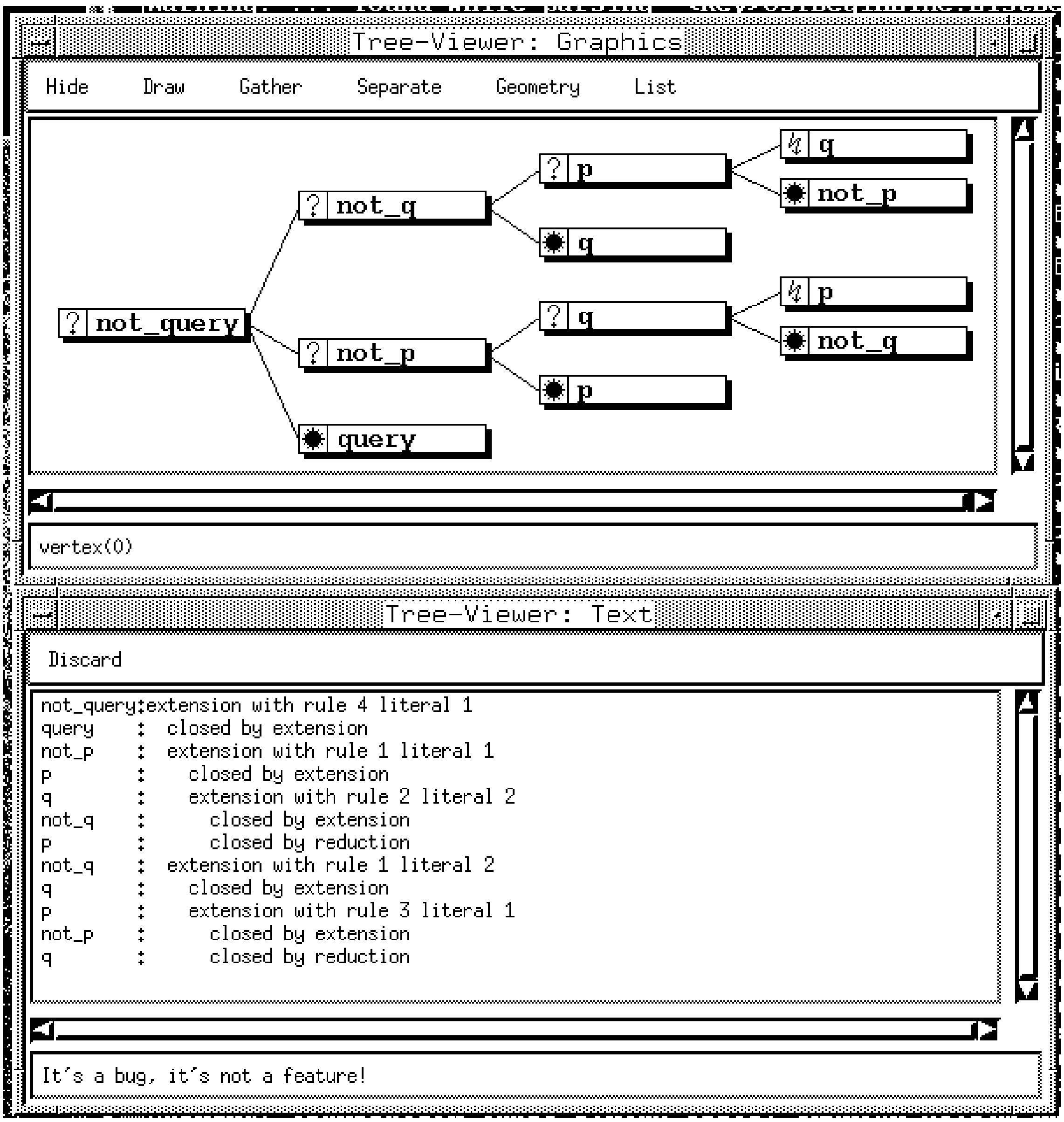


Figure 10: refutation tree presented by tree viewer tview

```
--------------------     statistics     --------------------

Proof Statistics
----------------
Proof Time:  0.033333 sec
Proof Depth: 3
Extension Steps: 4
Reduction Steps: 2              (incl. 2 with cut)
Factoring Steps: 0
Inferences:      8

--------------------      answers       --------------------
```

□

5 Example Input Files

Example 5.1 (all-for-two) Input file for Example 2.4:

```
protein_flag(mode,pl0).

 p ;  q.
 p ; -q.
-p ;  q.
-p ; -q.
```

□

Example 5.2 (MSC006-1) From the problem description in [TPTP]:
"Suppose there are two relations P and Q. P is transitive, and Q is both transitive and symmetric. Suppose further the "squareness" of P and Q: any two things are related either in the P manner or the Q manner. Prove that either P is total or Q is total."

```
% BEGIN_AXIOMS
% NAME: transitive(p)
p(A, C) :- p(A, B), p(B, C).

% NAME: transitive(q)
q(A, C) :- q(A, B), q(B, C).

% NAME: symmetry(q)
q(B, A) :- q(A, B).

% Name: all_related(p,q)
p(A, B) ; q(A, B).

% NAME: not_totality(q)
-p(a, b).

% NAME: goal
?- q(c, d).
% END_AXIOMS
```

□

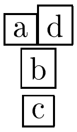
Example 5.3 (MSC006-th) Automatically created theory file for 5.2:

```
% +++ linear completion of axiomset : trans +++
% NAME: transitive(p)
% THEORY: [p(X, Z) :- (p(X, Y), p(Y, Z))]
```

```
p(A, B), p(B, C) -> p(A, C).
~ p(A, C), p(B, C) -> ~ p(A, B).
% NAME_END

% +++ linear completion of axiomset : trans +++
% NAME: transitive(q)
% THEORY: [q(X, Z) :- (q(X, Y), q(Y, Z))]
q(A, B), q(B, C) -> q(A, C).
q(C, B), ~ q(A, C) -> ~ q(A, B).
~ q(C, A), q(B, C) -> ~ q(A, B).
% NAME_END

% +++ linear completion of axiomset : sym +++
% NAME: symetric(q)
% THEORY: [q(X, Y) :- q(Y, X)]
q(A, B) -> q(B, A).
~ q(B, A) -> ~ q(A, B).
% NAME_END

end(theory).

% BEGIN_AXIOMS
% NAME: all_related(p,q)
p(A, B) ; q(A, B).

% NAME: not_totality(q)
~p(a,b).

% NAME: goal
?- q(c,d).
% END_AXIOMS
```

□

Example 5.4 (blocks world) This example was invented by Peter Baumgartner. Consider the following situation: On a blue block c is situated a block b which is blue or green, on which two green blocks a and d are placed.




```
protein_flag(answers,more).

on(a,b).
on(d,b).
on(b,c).
```

```
colour(a,green).
colour(d,green).
colour(b,blue) ; colour(b,green).
colour(c,blue).

?- protein_answer(on(X,Y)), on(X,Y), colour(X,green), colour(Y,blue).
```

□

Example 5.5 (knights and knaves) This example is taken from [Smu78]. It is examined in [BauFurSto95].
On an island, there live exactly two types of people: knights and knaves. Knights always tell the truth and knaves always lie. I landed on the island, met two inhabitants, asked one of them: "Is one of you a knight?" and he answered me "no". What can be said about the types of the asked and the other person?

```
protein_flag(calculus,rme).
protein_flag(ancestry_rme,on).
protein_flag(definite_answers, on).
protein_flag(query_reuse, off).
```

% $true(isa(Q, knight)) \veebar true(isa(Q, knave))$

```
 true(isa(Q,knight)) ; true(isa(Q,knave)).
-true(isa(Q,knight)) ; true(isa(Q,knave)).
```

% $says(P, S) \rightarrow (true(S) \leftrightarrow true(isa(P, knight)))$

```
true(S) :- true(isa(P,knight)), says(P,S).
true(isa(P,knight)) :- true(S), says(P,S).
```

% $says(asked, not(or(isa(asked, knight), isa(other, knight))))$

```
says(asked,not(or(isa(asked,knight),isa(other,knight)))).
```

% $\neg true(isa(asked, X)) \vee \neg true(isa(other, Y))$

```
?- protein_answer(X/Y), true(isa(asked,X)), true(isa(other,Y)).
```

% $true(C) \veebar true(not(C))$

```
 true(C) ;  true(not(C)).
-true(C) ; -true(not(C)).
```

% $true(or(A, B)) \leftrightarrow (true(A) \vee true(B))$

```
true(A) ; true(B) :- true(or(A,B)).
true(or(A,B)) :- true(A).
true(or(A,B)) :- true(B).
```

□

Example 5.6 (contraints) This example uses the ECLiPSe constraint solving library to include constraints mechanism into PROTEIN using the `prolog`-environment.

```
prolog_pred(::/2).
prolog_pred(dvar_attribute/2).
prolog_pred(use_module/1).
prolog_pred(own_answer/1).
prolog_pred(writeln/1).

begin(prolog).
:- lib(fd).
own_answer(X) :- (dvar_attribute(X,D);true), X::[1..100], writeln(D), !.
end(prolog).

p(X) :- X::[1..100].

?- use_module(fd), p(X), own_answer(X).
```

□

References

[Bau96] P. Baumgartner · *Theory Reasoning in Connection Calculi and the Linearizing Completion Approach* · 1996, in preparation · URL = http://www.uni-koblenz.de/˜ peter/thesis.html

[BauBrü96] P. Baumgartner, S. Brüning · *A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion* · Journal of Automated Reasoning, 1996, to appear

[BauFurNie96] P. Baumgartner, U. Furbach, I. Niemelä · *Hyper Tableaux* · Research Report 8/96, University of Koblenz

[BauFurSto95] P. Baumgartner, U. Furbach, F. Stolzenburg · *Model Elimination, Logic Programming and Computing Answers* · proceedings of the International Joint Conference on Artificial Intelligence, Montreal, IJCAI 95, pages 1:335-340, 1995

[BauFur94a] P. Baumgartner and U. Furbach · *Consolution as a Framework for Comparing Calculi* · Journal of Symbolic Computation, 16(5), Academic Press, 1993

[BauFur94b] P. Baumgartner and U. Furbach · *Model Elimination without Contrapositives and its Application to PTTP* · A. Bundy (editor) Automated Deduction, CADE-12, LNAI 814, Springer, 1994

[BauFur94c] P. Baumgartner and U. Furbach · *PROTEIN: A PROver with a Theory Extension Interface* · A. Bundy (editor) Automated Deduction, CADE-12, pages 769-773, LNAI 814, Springer, 1994

[BauSto94] P. Baumgartner and F. Stolzenburg · *Contraint Model Elimination and a PTTP Implementation* · P. Baumgartner, R. Hähnle, J. Posegga, editors, proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods, TABLEAUX-4, pages 201-216, LNAI 918, Springer, 1995

[Bau93a] P. Baumgartner · *Refinements of Theory Model Elimination and a Variant without Contrapositives* · A. Cohn, editor, 11th European Conference on Artificial Intelligence, ECAI 94, Wiley, 1994 · long version: Research Report 8/93, University of Koblenz

[Bau93b] P. Baumgartner · *Linear Completion: Combining the Linear and the Unit-Resulting Restrictions* · Research Report 9/93, University of Koblenz

[Bau92] P. Baumgartner · *A Model Elimination Calculus with Built-in Theories* · H.-J. Ohlbach, editor, proceedings of the 16th German AI-Conference, GWAI 92, pages 30-42, Springer, 1992

[Lov68] D. Loveland · *Mechanical Theorem Proving by Model Elimination* · JACM, 15(2)

[Smu78] R.M. Smullyan · *What is the name of this book? The riddle of Dracula and other logical puzzels* · Prentice-Hall, Englewood Cliffs, NJ, 1978

[Sti88] M. Stickel · *A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler* · Journal of Automated Reasoning, 4:353-380

[Sti85] M. Stickel · *Automated Deduction by Theory Resolution* · Journal of Automated Reasoning, 1:333-355

[Sti83] M. Stickel · *Theory Resolution: Building in Nonequational Theories* · SRI International Research Report Technical Note 286, Artificial Intelligence Center

[TPTP] G. Sutcliffe,C.Suttner,T.Yemenis · *The TPTP problem library* · A. Bundy (editor) Automated Deduction, CADE-12, Springer, 1994

[URL-ILF] URL of the ILF-server `http://www-irm.mathematik.hu-berlin.de/~ilf-serv/`

[URL-Uni] URL of the University of Koblenz `http://www.uni-koblenz.de`

[URL-PROTEIN] You can get PROTEIN, many of the papers, and much more on the following WWW-page: `http://www.uni-koblenz.de/ag-ki/protein.html`

Index

Changes

5.7.1996 : first version

23.10.1996 : including dynamic handling of additional arguments, comment standards for postprocessing, and the syntax definition of the internal proof tree term

4.11.1996 : including delayed restart model elimination calculus variant

still missing : minimal model reasoning