The leanTAP-FAQ:

Frequently Asked Questions about leanTAP

Bernhard Beckert & Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe, Germany
{beckert|posegga}@ira.uka.de
WWW: http://emmy.ira.uka.de/~posegga/leantap/

FAQ.tex,v 1.5 1994/08/04 12:26:12 beckert Exp

Contents

Preface

Due to the increasing interest in leanTAP we decided to write this document; it provides answers to the most frequent question people have about leanTAP. We assume that the reader is familiar with (Beckert & Posegga, 1994a) or, even better, with (Beckert & Posegga, 1994b)[1].

leanTAP (Beckert & Posegga, 1994a) is a complete and sound theorem prover for classical first-order logic based on free-variable semantic tableaux. The unique thing about leanTAP is that it is probably the smallest theorem prover around: The program consists of only about 12 lines of Prolog.

leanTAP is originally implemented in Sicstus prolog, but runs as well under Quintus Prolog (all warnings of the Quintus compiler can be ignored). It should also be easy to port leanTAP to other Prolog dialects. The source code is distributed free of charge.

We appreciate feedback. If you should have suggestions to improve this FAQ, criticism, or any other comments, please contact us. If should you do something interesting with leanTAP, please do also inform us as well. This could be modifying or extending the program, applying it to some domain, or whatever.

There is also a mailing list of people who are interested in hearing about future developments of leanTAP. Drop one of us a line if you want to be included.

Source code for the standard version of leanTAP

We will often refer to the source code of leanTAP in this paper. Therefore, we repeat the source code for the standard version of leanTAP here:

```
% Conjunction:
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).

% Disjunction:
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).

% Universal Quantification:
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).

% Closing Branches:
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).

% Extending Branches:
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```

The program is explained in detail in (Beckert & Posegga, 1994a); here is a very brief outline: The predicate

```
prove(Fml,UnExp,Lits,FreeV,VarLim)
```

[1] See Appendix A on how to obtain a copy.

succeeds if there is a closed tableau for the first-order formula bound to `Fml`. The proof proceeds by considering individual branches (from left to right) of a tableau; the parameters `Fml`, `UnExp`, and `Lits` represent the current branch: `Fml` is the formula being expanded, `UnExp` holds a list of formulæ not yet expanded, and `Lits` is a list of the literals present on the current branch. `FreeV` is a list of the free variables on the branch (these are Prolog variables, which might be bound to a term). The positive integer `VarLim` is used to initiate backtracking; it is an upper bound for the length of `FreeV`.

leanTAP uses Prolog syntax for first-order formulæ: atoms are Prolog terms, "`-`" is negation, "`;`" disjunction, and "`,`" conjunction. Universal quantification is expressed as `all(X,F)`, where `X` is a Prolog variable and `F` is the scope. Thus, a first-order formula is represented by a Prolog term (e.g., `(p(0),all(N,(-p(N);p(s(N)))))` stands for $p(0) \wedge (\forall n(\neg p(n) \vee p(s(n))))$.

Questions and Answers

Q 1 How can I typeset "leanTAP" in TEX?

The official version is

```
\mbox{{\sf lean}$T^{\!\!\textstyle A}\!\!P$}
```

Q 2 How can leanTAP handle more logical connectives?

I would like to extend the program such that I can use additional logical connectives in formulæ. Which is the best way for doing this?

There are two principle ways for this: you can either extend the program for deriving negation normal form, or the leanTAP program itself. The former is very straightforward; look at the source code and you will see how to do it.

Extending leanTAP is also not very complicated. The way to do it is to write a new clause which handles the new connective according to its tableau rule. Two important points must be kept in mind:

1. New clauses for connectives *must* be placed *before* the last two clauses (cf. Question Q 11).
2. If your connective introduces negation (like equivalences would do), you must extend leanTAP further and include clauses for negated connectives as well.

Note, that 2. is a good argument for extending the NNF algorithm instead of leanTAP — unless you do want to get rid of the separate NNF derivation and include it into leanTAP, anyway.

Q 3 I have heard that leanTAP has been verified. Is this true?

Yes, leanTAP has been verified in the sense that it was formally proven (by hand) that

1. If the query `prove(`*fml*`,`*h*`,[],[],`*d*`)` to the program leanTAP returns *success* as an answer, where *fml* is a formula, *h* is a list of formulæ and *d* is a natural number, than $\neg fml$ is a logical consequence of *h*.
2. If $\neg fml$ is a logical consequence of *h* then there is a natural number *d* such that the query `prove(`*fml*`,`*h*`,[],[],`*d*`)` to leanTAP terminates with success.

For this, semantics of Prolog based on its computation tree have been used.[2] A draft of a paper including the verification of leanTAP is available upon request.

[2]Note, that the only occurrence of negation in the program (line 7) is not critical w.r.t. the semantics of Prolog. Cf. Question Q 15.

Q 4 Can leanTAP be programmed declaratively?

I have heard that Prolog should be used declaratively. leanTAP *is not particularly declarative. Can it be modified in this way?*

Well, it is at least arguable whether Prolog is declarative, or should be used declaratively. Anyway, leanTAP can be programmed declaratively, although it will probably loose some of its efficiency.

For re-programming leanTAP declaratively, we first get rid of all explicit cuts in the first three clauses. We should then ensure that the last two clauses are entered with literals as the first argument, only. This can be done by ensuring that the principle functor is not a binary logical connective.

The last but one clause for closing branches will probably need further revision if we want to achieve reasonably efficient, declarative code: the implicit implementation of *member* should be removed and programmed explicitly. The same applies to the implicit cut. Refer to Question Q 12 for details.

Q 5 Is there a way to add equality?

I would like to use leanTAP *for proving formulæ from first order logic* with equality*. Due to the huge search space, it is not a practical solution just to add the equality axioms to the input. How can* leanTAP *be extended to efficiently handle equality?*

There is a simple answer to this question: Use E-unification to close branches instead of syntactical unification. In practice, however, it is not as easy.

To handle equality in free variable tableaux, it is not sufficient to solve classical E-unification problems, where the equalities are (implicitly) universally quantified; instead, *rigid* E-unification problems have to be solved, where in an equational proof the equalities can be used with only one instantiation for the free variables they contain. That is, given a set E of equalities, and terms s and t, substitutions σ have to be found, such that

$$E\sigma \models^{\circ} (s\sigma \approx t\sigma)$$

(see Question Q 7 for the definition of $\models^{\circ}$).

If a predicate `rigid_e_unify(`E`,`s`,`t`)` is provided that implements rigid E-unification, i.e., that enumerates the rigid E-unifiers of s and t in a complete way, then it is very easy to add the handling of equality to leanTAP. Unfortunately, we do not know of an sufficiently efficient implementation that would be short enough to print it here and that, thus, would be in accordance with the philosophy of lean deduction.

However, there is a "complex" implementation of the completion-based method for solving rigid E-unification problems described in (Beckert, 1994). The source code is available from the authors.

Q 6 Can leanTAP be extended to non-classical logics?

As long as the language underlying a logic is semi-decidable and a complete and sound tableau calculus is known, it should be possible to adapt leanTAP for deduction in this calculus.

More concretely, we know of people who thought about it, but, at least to our knowledge, there is no running program so far. If you should succeed, let us know.

Q 7 Are input formulæ required to be closed?

All standard first-order test examples for theorem provers are closed formulæ, i.e.: they do not contain free variables. Contrary to resolution where all clauses are implicitly universally quantified, a free-variable tableau calculus can also be used on formulæ with free variables. What does

leanTAP *do in this case?*

This is a very fine logical point; the question (and the answer) is probably not easy to understand. First, we must clarify what is meant by "logical consequence". Two versions appear in the literature:

1. $A \models B$, i.e., for each interpretation $\mathbf{I}$: if $\mathrm{val}_{\mathbf{I},\beta}(A) = \mathit{true}$ for each variable assignment β, then $\mathrm{val}_{\mathbf{I},\beta'}(B) = \mathit{true}$ for each variable assignment β';
2. $A \models^{o} B$, i.e., for each variable assignment β and for each interpretation $\mathbf{I}$: if $\mathrm{val}_{\mathbf{I},\beta}(A) = \mathit{true}$, then $\mathrm{val}_{\mathbf{I},\beta}(B) = \mathit{true}$.

The latter is often called *strong consequence relation*. Both do not differ for closed formulæ A and B. For other formulæ there is a difference; here is an example: $p(x) \models \forall x\, p(x)$, but $p(x) \not\models^{o} \forall x\, p(x)$.

The strong consequence relation is usually underlying tableau calculi. In this sense, leanTAP behaves correctly: `prove(Fml,...)` "implements" `fml` $\models^{o}$ *false*.

However, the Skolemization we use is not correct in this sense: $p(x) \wedge \neg(\forall y\, p(y))$ will be Skolemized to $p(x) \wedge \neg p(p(y))$; but $p(x) \wedge \neg(\forall y\, p(y)) \not\models^{o}$ *false* and $p(x) \wedge \neg p(p(y)) \models^{o}$ *false*.

Q 8 Don't you need the quantified variables in input formulæ to be disjoint?

leanTAP *uses first-order variables as Prolog variables, and vice versa. Thus, the same variable in different scopes (like* `all(X,p(X))` *and* `all(X,q(X))`*) is likely to cause problems.*

Good thinking. However, we took this into account: the original variable appearing in the input formula is actually never used in a proof. A quantified formula (and the corresponding quantified variable) is only stored in the list `UnExp` of unexpanded formulæ. Before considering it in the proof, the variable is renamed. Therefore, this does not cause problems.

Q 9 How important is the sequence of arguments in `prove`?

It does not matter at all if you neglect efficiency. For efficiency, it is important that the right clause of the program is chosen at the right time. As Prolog systems usually perform indexing for selecting clauses based on the first argument, the current formula should be at this place. The positions of other arguments of `prove` do not matter much.

Q 10 Can I also quantify over more than one variable?

It would be much more convenient to write something like `all([X,Y],p(X,Y))` *rather than* `all(X,all(Y,p(X,Y)))`*. How can I achieve this?*

Just do it. It works. No need to change anything.

But note, that the limit `VarLim` for controlling backtracking has different semantics then: in this case, it does not limit the number of free variables introduced on a branch, but the number of applications of γ-rules. However, this should not matter much.

Q 11 What happens if the clauses are reordered?

Just out of curiosity: What happens if I change the sequence of Prolog clauses in the program?

This is a nice puzzle. Let us have a look at some possibilities:

The first three clauses can be arbitrarily interchanged — their position in the program (relative to each other) is not significant.

If clause four and five are exchanged, things become more complicated: In this case, the prover will not close branches before the tableau has been fully expanded up to the given limit

`VarLim`. Thus, leanTAP remains complete, but becomes inefficient. This can be seen as a general rule: *Put the clause for closing branches before the one for extending, as it is reasonable to close branches as early as possible.*

Now, let us put clause four (for closing branches) at the beginning of the program: In this case, leanTAP would always try first to close a branch — before expanding the formulæ on it. It would result in the possibility to clause a branch with complex formulæ as well, and not only on the level of literals as the original leanTAP does. In fact, this is the standard use in tableau calculi. However, as leanTAP is assumed to work on negation normal form, it can never happen: there will be no complementary formulæ on the branches besides literals. Thus, putting clause four at the beginning would cause an unnecessary overhead.

As a last possibility in this game, we consider putting the last clause at the beginning. This causes again inefficient behavior: leanTAP will put every formula into the list of literals, first. Then, it will be taken out again and expanded.

Q 12 Can you please explain the clause in line 11 again?

The fourth clause for closing branches seems to be the most obscure one. I do not quite understand what is going on, here.

Well, there is a trick here: we implicitly implemented a `member` predicate. A probably more readable version of this clause would be:

```
prove(Lit,_,Lits,_,_) :-
  (Lit = -Neg; -Lit = Neg) -> memberunify(Neg,Lits).
```

where

```
memberunify(Element,[Head|Tail]) :-
  (unify(Element,Head) ; memberunify(Element,Tail)).
```

When designing leanTAP, we decided not to use a separate clause but encode leanTAP with a single predicate. It is a matter of taste which version one prefers.

Q 13.1 What happens if I replace the implication in line 12 by a cut?

It appears to me that replacing

```
(Lit = -Neg; -Lit = Neg) -> ...
```

by

```
(Lit = -Neg; -Lit = Neg),!, ...
```

is better Prolog. Would it be the same?

It clearly results in incompleteness: the implication "`->`" is sort of an implicit cut; it does not affect the Prolog search tree at the level of the clause where the implication occurs. An explicit cut as proposed would let leanTAP fail as soon as the unification that follows fails once.

Q 13.2 I see. But how about leaving the cut out?

Thus, you use "`(Lit = -Neg; -Lit = Neg), ...`*" instead.*

This will work, but it will generate double negation during backtracking. It does not hurt, but results in overhead.

Q 14 Why is the γ-formula in line 9 put at the end of the list?

I understand that you keep universally quantified formulæ in the list of unexpanded formulæ, since they may be used more than once. Why do you store them at the end of the list, rather than at the beginning? This would save you a call to the `append` *predicate and should be more efficient.*

This is a good observation, but it does not work. Backtracking in leanTAP is controlled by restricting the number of free variables that may be introduced on a branch. It does not matter which γ-formulæ these variables come from. Therefore, you must ensure that each γ-formula has a chance of being expanded. This is the reason for using a queue and not a stack. The latter would cause that the same γ-formula is expanded over and over again and all others would be ignored. leanTAP would be incomplete.

Q 15 What is the actual purpose of line 7?

Literally, the goal in line 7 succeeds if the number of free variables on the current branch is not equal to `VarLim`. Actually, it functions as a *less-than* expression: as used, it is equivalent to

```
length(FreeV,L), L < VarLim.
```

Q 16 Is `FreeV` really a list of the free variables on the current branch?

No, not exactly. The correct characterization is: `FreeV` is a list of the Prolog terms that the free variables which have been introduced on the current branch are bound to. Therefore, `FreeV` can contain (ground) terms, free variables that have been introduced on other branches, and even variables that do not occurr in the anywhere tableau (these stem from applying the γ-rule to formulæ like $(\forall x)p(y)$).

However, for the correctness of leanTAP, as well as for the understanding of how is works, this does not make any difference.

Q 17.1 Is it really correct to use a formula to Skolemize itself?

In the predicate `nnf/2` *for computing negation normal form, that comes with* leanTAP*, to Skolemize a formula* $\Phi = (\exists x)\phi(x)$ *you use* $\phi(x)$ *as a Skolem term. Is this really correct?*

It is correct, provided the sets of predicate and function symbols are disjoint. In (Beckert *et al.*, 1993) it has been proven to be correct to use a Skolem term that (i) contains all the free variables occurring in the formula Φ to be Skolemized, and that (ii) is unique to the class $[\Phi]$ of formulæ that are identical to Φ up to variable renaming. Thus, if $y_1, \ldots, y_n$ are the free variables in Φ, we could use the Skolem term $f_{[\Phi]}(y_1, \ldots, y_n)$; the result of the Skolemization then would be $\phi(f_{[\Phi]}(y_1, \ldots, y_n))$. However, to construct this term, a new symbol has to be generated and the free variables have to be extracted from Φ.

Instead, we can use the skope $\phi(x)$ for Skolemization: It is unique to $[\Phi]$ (up to variable renaming), and it contains the free variables occurring in Φ (the additional variable x does not do any harm, because it is never instantiated). Then, the result of Skolemizing the formula $(\exists x)\phi(x)$ is $\phi(\phi(x))$.

Q 17.2 But don't you get a cyclic term?

I see. But doesn't Skolemizing this way result in a cyclic term?

The result is not cyclic, because the Prolog variable representing x is *substituted* by the Prolog term representing $\phi(x)$ (and not bound to it).

A How to get the source code and papers on leanTAP

The easiest way to access the material on leanTAP is opening the page

`http://emmy.ira.uka.de/~posegga/leantap/`

on the *World Wide Web*. With this document you can retrieve the program and the corresponding papers online.

For those who have no access to this, we describe the access by anonymous ftp in the sequel. If you should not have ftp access, either, contact the authors.

The current reference for leanTAP is (Beckert & Posegga, 1994a). This describes the basic version of leanTAP. Besides this, there is an enhanced version of leanTAP which includes a powerful heuristic called "universal formulæ". Both programs are described in the long version of the paper (Beckert & Posegga, 1994b), which is currently under review. We recommend that you read the long paper, rather than the one cited above.

The long paper (Beckert & Posegga, 1994b) and the source code for leanTAP are available via anonymous ftp on Internet from `sonja.ira.uka.de` (129.13.31.3). Open this host with an ftp-program, log in as "anonymous" and type your email address as password. `cd` to the directory `pub/posegga` and switch to binary file transfer mode. This is usually done by typing `binary` in your ftp program.

The paper lives in the file `LeanTaP.ps.Z`, which is a compressed Postscript file[3]. The source code is stored as a compressed shell archive in `LeanTaPsrc.shar.Z`.

Both files need to be uncompressed first. Under Unix, this works with the shell command

`% uncompress` *filename*

Whilst the Postscript file can be printed then then, the source code in the shell archive must be unpacked:

```
% sh LeanTaPsrc.shar
x - extracting README (Text)
x - extracting leantap.pl (Text)
x - extracting leantest.pl (Text)
x - extracting nnf.pl (Text)
x - extracting unify.pl (Text)
```

The Prolog code in the file `leantest.pl` defines four predicates which are supposed to be something like a user interface:

`provefml/1`	the standard version of leanTAP
`incprovefml/1`	the standard version using iterative deepening
`uv_provefml/1`	the version with universal variables
`uv_incprovefml/1`	universal variables and iterative deepening.

All these predicates get one argument, which is the name of a formula in the database. leanTAP comes with a database of some of Pelletier's problems (Pelletier, 1986) in `leantest.pl`.

The actual prover lives in the file `leantap.pl` and is defined as the predicates `prove/2` and `prove_uv/2`. See the comments there for details. The documentation for the code is the paper mentioned above.

The file `nnf.pl` contains a program for deriving Skolemized negation normal form, used by `leantest.pl`. Again, refer to the paper on how `nnf.pl` works.

Depending on the concrete Prolog system you intend to use, you will, or will not need the code in `unify.pl`: it contains a predicate `unify/2` which performs sound unification, used by the prover in `leantap.pl`. If your Prolog system has built-in sound unification, you do not need this file if you change the call to `unify` in `leantap.pl` appropriately.

[3]You need a printer capable of understanding Postscript-II to print it.

B Running leanTAP: an example

```
629.dao % sicstus
SICStus 2.1 #8: Mon Aug 30 15:43:08 MET DST 1993
| ?- compile(leantest).
{compiling /home/emmy/posegga/tmp/leantest.pl...}
{compiling /home/emmy/posegga/tmp/leantap.pl...}
{loading /tools/sicstus2.1/library/lists.ql...}
{loaded /tools/sicstus2.1/library/lists.ql in module lists, 60 msec 31888 bytes}
{compiling /home/emmy/posegga/tmp/unify.pl...}
{compiled /home/emmy/posegga/tmp/unify.pl in module unify, 230 msec 11056 bytes}
{compiled /home/emmy/posegga/tmp/leantap.pl in module leantap, 630 msec 55360 bytes}
{compiling /home/emmy/posegga/tmp/nnf.pl...}
{compiled /home/emmy/posegga/tmp/nnf.pl in module nnf, 190 msec 9680 bytes}
{compiled /home/emmy/posegga/tmp/leantest.pl in module leantest, 1820 msec 101040 bytes}

yes
| ?- provefml(pel28).
pel28 proved in 0 msec, VarLim = 3

yes
| ?- incprovefml(pel28).
pel28  proved in 10 msec, found VarLim = 3

yes
| ?- uv_provefml(pel28).
pel28 proved in 10 msec, VarLim = 3

yes
| ?- uv_incprovefml(pel28).
pel28  proved in 9 msec, found VarLim = 3

yes
| ?- uv_provefml(X),fail.
pel1 proved in 0 msec, VarLim = 0
pel2 proved in 0 msec, VarLim = 0
pel3 proved in 0 msec, VarLim = 0
pel4 proved in 0 msec, VarLim = 0
pel5 proved in 0 msec, VarLim = 0
pel6 proved in 0 msec, VarLim = 0
pel7 proved in 0 msec, VarLim = 0
pel8 proved in 10 msec, VarLim = 0
pel9 proved in 0 msec, VarLim = 0
pel10 proved in 0 msec, VarLim = 0
pel11 proved in 0 msec, VarLim = 0
pel12 proved in 10 msec, VarLim = 0
pel13 proved in 0 msec, VarLim = 0
pel14 proved in 0 msec, VarLim = 0
pel15 proved in 0 msec, VarLim = 0
pel16 proved in 0 msec, VarLim = 0
pel17 proved in 0 msec, VarLim = 0
pel18 proved in 0 msec, VarLim = 2
pel19 proved in 0 msec, VarLim = 2
pel20 proved in 9 msec, VarLim = 6
pel21 proved in 0 msec, VarLim = 2
pel22 proved in 0 msec, VarLim = 2
pel23 proved in 0 msec, VarLim = 1
pel24 proved in 30 msec, VarLim = 6
```

```
pel25 proved in 0 msec, VarLim = 3
pel26 proved in 10 msec, VarLim = 3
pel27 proved in 9 msec, VarLim = 4
pel28 proved in 10 msec, VarLim = 3
pel29 proved in 9 msec, VarLim = 2
pel30 proved in 10 msec, VarLim = 2
pel31 proved in 9 msec, VarLim = 3
pel32 proved in 10 msec, VarLim = 3
pel33 proved in 9 msec, VarLim = 1
pel34 proved in 109 msec, VarLim = 5
pel35 proved in 0 msec, VarLim = 4
pel36 proved in 0 msec, VarLim = 6
pel37 proved in 20 msec, VarLim = 7
pel38 proved in 339 msec, VarLim = 4
pel39 proved in 10 msec, VarLim = 1
pel40 proved in 9 msec, VarLim = 3
pel41 proved in 0 msec, VarLim = 3
pel42 proved in 10 msec, VarLim = 3
pel43 proved in 109 msec, VarLim = 5
pel44 proved in 10 msec, VarLim = 3
pel45 proved in 40 msec, VarLim = 5
pel46 proved in 100 msec, VarLim = 5

no
| ?-
```

References

Beckert, Bernhard. 1994. A Completion-Based Method for Mixed Universal and Rigid E-Unification. *Pages 678–692 of:* Bundy, A. (ed), *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France.* LNCS 814. Springer.

Beckert, Bernhard, & Posegga, Joachim. 1994b. leanT^AP*: Lean Tableau-based Deduction.* Submitted.

Beckert, Bernhard, & Posegga, Joachim. 1994a. leanT^AP: Lean Tableau-Based Theorem Proving. Extended Abstract. *Pages 793–797 of:* Bundy, A. (ed), *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France.* LNCS 814. Springer.

Beckert, Bernhard, Hähnle, Reiner, & Schmitt, Peter H. 1993. The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux. *Pages 108–119 of:* Gottlob, G., Leitsch, A., & Mundici, D. (eds), *Proceedings, 3rd Kurt Gödel Colloquium (KGC), Brno, Czech Republic.* LNCS 713. Springer.

Pelletier, Francis Jeffry. 1986. Seventy-Five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, **2**, 191–216.