A Time Efficient KE Based Theorem Prover

Ulrich Endriss

Department of Computer Science, King's College London,
Strand, London WC2R 2LS, UK, Email: `endriss@dcs.kcl.ac.uk`
URL: `http://www.dcs.kcl.ac.uk/~endriss/`

Abstract. We present a proof procedure based on the KE calculus for propositional logic and its implementation as a short Prolog program. The procedure's time complexity is discussed and compared to that of an efficient Tableau based prover.

1 Introduction

The KE calculus [2] is a refutation system close to Tableau. The crucial feature that distinguishes the former from the latter is the integration of an analytic cut rule (*PB*). Even though some sort of 'superiority' of KE over Tableau in terms of proof size has been stated in the literature, until now no *implemented* KE based proof procedure can compete with state-of-the-art Tableau provers as far as runtimes are concerned.

The aim of this work has been to close that gap, at least for the case of classical propositional logic. As a benchmark we take leanT^AP [1], a 'lean' Tableau based theorem prover implemented in Prolog, which is simple and efficient. As leanT^AP was build for first order logic we will first reduce it to a propositional prover in order to guarantee a 'fair competition'. Then a KE based proof procedure is designed in a similar fashion. The problems naturally arising during such a transformation are addressed and – where possible – solved. We conclude with an experimental comparison of the two procedures.

2 Space and Time

In [2] it has been shown that KE linearly simulates the Tableau method, whereas the latter cannot p-simulate KE, in other words: *KE proofs are basically shorter than Tableau proofs.* This is in fact true – with respect to space – for 'ideal' (again, with respect to space) proof procedures.

But, from that observation alone, we *cannot* conclude, that for a specific problem the KE deduction is also *faster* than the Tableau deduction. Apart from the space complexity results also the following points need to be considered:

- The time taken by a proof procedure depends on the number of derived formulae *and* on the time required to derive one such formula. In a KE based procedure an application of a beta rule takes much more time than any step (apart from closing a branch) in a Tableau prover.

– KE has more rules than Tableau. A proof procedure has to check which rule to apply to a given formula. As there are fewer possibilities in Tableau, the checking will be faster in that setting.
– A fast prover is not necessarily ideal with respect to space. leanT^AP for instance does not build up a minimal proof tree, but still is very time efficient.

3 A Tableau Procedure for Propositional Logic

The Prolog program leanT^AP as defined in [1] implements a small theorem prover for first order logic, based on free-variable semantic tableaux. Table 1 shows an adaptation for propositional logic, which we call `tap`.[1] Like the original, `tap` is restricted to negation normal form (*NNF*), i.e. negation has to be pushed down to the atomic level before deduction starts.

```
tap( (A,B), Fmls, Lits) :- !,     % apply alpha
    tap( A, [B|Fmls], Lits).

tap( (A;B), Fmls, Lits) :- !,     % apply beta
    tap( A, Fmls, Lits), !,
    tap( B, Fmls, Lits).

tap( Lit, _, Lits) :-             % close branch
    (Lit = -(C) ; -(Lit) = C) -> member( C, Lits).

tap( Lit, [Fml|Fmls], Lits) :-   % next formula
    tap( Fml, Fmls, [Lit|Lits]).
```

Table 1. leanT^AP for propositional logic

To obtain `tap` from leanT^AP the clauses handling quantified formulae have been omitted and the clause for closing branches has been simplified as no occur check is necessary. Due to the nature of these simplifications, it is clear that `tap` will be slightly faster than leanT^AP for propositional logic.

4 Designing the KE Proof Procedure

In KE proof trees are constructed in a similar way to the Tableau method. Alpha rules and the notion of a closed branch are identical for both calculi. KE beta rules are linear and take two premises. For example from $A \vee B$ and $\neg A$ we can infer B. Unlike Tableau, KE is not cut-free. Following the principle of bivalence (*PB*) a branch may be split adding a formula A to the left and its negation $\neg A$ to the right branch. For analytic KE the choice for such *PB*-formulae is restricted to subformulae of beta formulae that are already on the branch to be split [2].

[1] The provers described in this paper can handle conjunction, disjunction, and negation, which have been represented in Prolog as ‘`,`’, ‘`;`’, and ‘`-`’, respectively.

When trying to follow the lines of `tap`'s design to construct a KE based theorem prover (for propositional formulae in *NNF*) we encounter the following problems:

- If *PB* is applied to a non-atomic subformula, its negation will not be in *NNF*. On the other hand we cannot simply restrict *PB* to literals, as the remaining calculus would not be complete.
- The most time consuming steps during proof search are those where you have to search a list for a matching formula, i.e. closure (both calculi) and KE's beta rules. For closing branches it is possible to restrict this search to complementary literals. It would be nice to have a similar restriction for the search of complements of minor premises for beta formulae. Unfortunately, KE is not complete if beta rules can only be applied to literals as minor premises.

To overcome those difficulties we introduce an adaptation of KE, which we will call KE*. Informally we obtain KE* from KE by restricting the application of beta rules to literals as minor premises, with one exception: directly after every application of *PB* the next (obvious) application of beta is performed in any case. For example, if *PB* is applied to A, the left subformula of $A \vee B$, then write A on the left branch, and $\neg A$ *and* B on the right one (whether A is a literal or not). KE* is restricted to formulae in *NNF*. The problem addressed before, namely that *PB* can produce non-*NNF* formulae is solved by immediately transforming the negated *PB*-formula into *NNF*. KE* is easily shown to be sound and complete (via a 'reduction' to KE and Tableau, respectively).

The simplest transformation of `tap` into a KE* proof procedure would only involve replacing the Tableau beta rule with the *two* beta rules for KE* (one for the left and one for the right subformula) and the new *PB* rule. This procedure can be improved by holding back beta formulae unless there are no more unexpanded alpha formulae on the branch.

A Prolog implementation of this procedure, which we call `kep`, is shown in Table 2. The alpha rule is the same as for `tap`. So is the clause which moves the active literal into the lists `Lits` and puts the next unexpanded formula into focus ('next formula'). The second clause does the 'storing' of beta formulae: they are temporarily stored in the list `Betas` and the next formula is tackled. The last clause takes the first element of that list of beta formulae and puts it into focus, if there are no more unexpanded formulae left in the main list `Fmls`. Also the implementation of the beta rule for the left subformula is straightforward. An attempt to apply beta is only made if the left subformula `A` is a literal. For the second beta rule things are more complicated. If the left subformula `A` is also a literal, we already know that the complement of `A` is *not* on the branch (i.e. in `Lits`). Otherwise beta would have been applied to it before. Because the conclusion of the beta rule is `A`, the next step would be to try to close the branch using `A`, i.e. to search `Lits` again. As we do not want to repeat this time consuming search, which is bound to fail anyway, that step can be omitted; `A` can be added to the list of literals directly, and the next formula can be addressed. If there is no such formula left, the procedure fails, because the branch cannot

```
kep( (A,B), Fmls, Betas, Lits) :- !,       % alpha
  kep( A, [B|Fmls], Betas, Lits).

kep( (A;B), [Fml|Fmls], Betas, Lits) :- !, % store beta
  kep( Fml, Fmls, [(A;B)|Betas], Lits).

kep( (A;B), [], Betas, Lits) :-            % apply beta: left
  (literal( A) -> ((A = -(C); -(A) = C) -> member( C, Lits))), !,
  kep( B, [], Betas, Lits).

kep( (A;B), [], Betas, Lits) :-            % apply beta: right
  (literal( B) -> ((B = -(C); -(B) = C) -> member( C, Lits))), !,
  (literal( A)
  -> Betas = [Beta|Rest], kep( Beta, [], Rest, [A|Lits])
  ;  kep( A, [], Betas, Lits)).

kep( (A;B), [], Betas, Lits) :- !,         % apply pb
  (literal( A)
  -> Betas = [Beta|Rest], kep( Beta, [], Rest, [A|Lits])
  ;  kep( A, [], Betas, Lits)),
  nnf( -(A), NNF), !,
  (literal( B)
  -> kep( NNF, [], Betas, [B|Lits])
  ;  kep( NNF, [B], Betas, Lits)).

kep( Lit, _, _, Lits) :-                   % close branch
  (Lit = -(C); -(Lit) = C) -> member( C, Lits).

kep( Lit, [Fml|Fmls], Betas, Lits) :- !,   % next formula
  kep( Fml, Fmls, Betas, [Lit|Lits]).

kep( Lit, [], [Beta|Betas], Lits) :-       % next beta formula
  kep( Beta, [], Betas, [Lit|Lits]).
```

Table 2. The KE based theorem prover `kep` for propositional logic

be closed. Similarly, when applying *PB* we already know, that, if one of the subformulae is a literal, its complement will not be found in `Lits`. So again, time can be saved. Note that the negated *PB*-formula is directly transformed into *NNF*.

5 Performance: Tableau v. KE

Applying a beta rule in KE reduces the number of branches compared to Tableau, but in Tableau such additional branches can be closed directly after having applied beta. For the given procedures those two actions have the same time complexity. In `tap` we have one basic step for the application of beta and one search through the list of literals for the closure. For `kep` we first search the list for the complement of the literal subformula and then we have the basic step of the actual rule application. What remains are four major differences between the two procedures that determine which of them will perform better when trying to refute a set of formulae.

- As `kep` has more clauses than `tap`, for every formula on the tree more checks of which clause applies have to be made.
- As in Prolog it is easier to insert an element at the beginning of a list than at the end, the storing of beta formulae in `kep` changes the order in which the two procedures analyse those formulae. What impact this has on the proof size is not clear and depends very much on the specific example.
- *PB* introduces a new formula on the right branch, the negated *PB*-formula, which Tableau does not do. This may help closing a branch earlier, but could also distract from applying the 'right' rules.
- For `kep` the transformation into *NNF* during an application of *PB* will require additional time.

	`tap`			`kep`		
No.	Time ($msecs$)	Formulae Derived	Branches Closed	Time ($msecs$)	Formulae Derived	Branches Closed
1	4	16	4	4	13	2
2	2	6	2	2	7	2
3	2	6	1	2	6	1
4	4	16	4	4	13	2
5	4	18	3	4	12	2
6	1	2	1	1	2	1
7	1	2	1	1	2	1
8	2	6	2	2	5	1
9	4	22	9	4	16	3
10	6	38	9	8	42	7
11	2	6	2	2	7	2
12	26	138	24	41	185	32
13	6	36	9	7	28	3
14	8	42	10	13	52	9
15	4	16	4	4	13	2
16	2	6	1	2	6	1
17	10	64	14	12	39	3

Table 3. Performances of `tap` and `kep` on the Pelletier Problems 1–17

Table 3 shows results for the runtimes of `tap` and `kep` on the Pelletier Problems for propositional logic [4]. Both programs have been tested on a Sun Sparc 10 running SWI-Prolog 2.1. The times given (average runtime for 100 tests) include the search for a *NNF*. While `kep` derives slightly fewer formulae and requires about three quarters of the branches,[2] it is on average around 10% slower than `tap`.

In [3] the KE based prover lean**KE** is compared with leanT^AP (both for first order logic). It derives slightly fewer formulae than `kep` and closes slightly fewer branches. The average runtime compared to leanT^AP on the same set of problems is around 350%. That leanT^AP is that much faster than lean**KE** is partly due to the size of the latter: lean**KE** has many more clauses, which means that for every formula to be analysed the time to find the right clause is longer. Moreover, lean**KE**, unlike `kep`, does not implement a strategy preventing it from searching for the complement of a formula a second time after a beta rule or *PB* has been applied.

Acknowledgements. The author would like to thank Marco Mondadori, Marcello D'Agostino, and Bernhard Beckert for many inspiring discussions, and two anonymous referees for their helpful comments.

References

1. B. Beckert and J. Posegga. leanT^AP: Lean Tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
2. M. D'Agostino and M. Mondadori. The taming of the cut. Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.
3. J. Pitt and J. Cunningham. Theorem proving and model building with the calculus KE. *Bulletin of the IGPL*, 4:129–150, 1995.
4. F. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[2] Remember that both procedures have not been designed for minimal space requirements. The smallest possible proof trees are smaller for most of the examples.