ileanTAP: An Intuitionistic Theorem Prover

Jens Otten*

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10, 64283 Darmstadt, Germany
jeotten@informatik.th-darmstadt.de

Abstract. We present a Prolog program that implements a sound and complete theorem prover for first-order intuitionistic logic. It is based on free-variable semantic tableaux extended by an additional string unification to ensure the particular restrictions in intuitionistic logic. Due to the modular treatment of the different logical connectives the implementation can easily be adapted to deal with other non-classical logics.

1 Introduction

Intuitionistic logic, due to its constructive nature, has an essential significance for the derivation of verifiably correct software. Unfortunately it is much more difficult to prove a theorem in intuitionistic logic than finding a classical proof for it. Whereas there are many classical provers there exists only very few (published) implementations of theorem provers for first-order intuitionistic logic (e.g. [8, 9]).

The following implementation was inspired by the classical prover leanTAP [1, 2]. leanTAP is based on free-variable tableaux [4], works for formulae in non-clausal form and reach its considerable performance by a very compact representation and an optimized Skolemization. To extend leanTAP to deal with intuitionistic logic one possibility is to add (or modify) the clauses implementing the corresponding tableau rules appropriately. This would of course not lead to a very efficient implementation, since the non-permutabilities between certain intuitionistic rules cause a large search space. A lot of additional strategies have to be added in this case (as done in [8]).

The following approach solves this problem in a more sophisticated way. In classical provers usually *term unification* and Skolemization is used to express the non-permutabilities between the quantifier rules (due to the "eigenvariable condition" in the sequent calculus). To handle the non-permutabilities between certain intuitionistic rules in a similar way we use a specialed *string unification* and extend the Skolemization accordingly. The basis of this approach was invented by Wallen who developed a matrix characterization for some non-classical logics [10].

In the following implementation we first use a leanTAP like technique for *path checking* to prove the *classical* validity of a given formula. Afterwards we try to unify the so-called *prefixes* of those atoms closing the branches of the tableau proof found in the first step. If this additional string unification succeeds the formula is *intuitionistically* valid. We present some performance results and show how to modify the code to deal with other non-classical logics.

The source code of ileanTAP can be obtained free of charge from the author.[2]

* The author is supported by the Adolf Messer Stiftung

[2] Or via web http://aida.intellektik.informatik.th-darmstadt.de/~jeotten/ileantap/

2 The Program

We assume the reader to be familiar with free-variable tableaux [4] and the leanTAP code (see [1, 2]) as well as with some details of Wallen's approach (see [10] or [7, 5]). The following Prolog implementation is of course not as lean and compact as the original code of leanTAP. The logical connectives and quantifiers of intuitionistic logic need a separate treatment and we can not make use of any negation normal form.[3] We divide the description of the implementation into the parts "path checking" and "T-string unification". Whenever possible we will use the notation of leanTAP.[4] For the syntax of formulae we use the logical connectives "`~`" (negation), "`,`" (conjunction), "`;`" (disjunction), "`=>`" (implication), "`<=>`" (equivalence), the quantifiers `all X:F` (universal) and `ex X:F` (existential), and Prolog terms for atomic formulae. For example to express the formula $\forall x \exists y(\neg q \wedge p(y) \Rightarrow p(x))$ we use the Prolog term `all X:ex Y:(~q,p(Y)=>p(X))`.

2.1 Path Checking

The technique of path checking is similar to the one used in leanTAP. In order to get a compact code and to allow an easy adaptation to other logics (see conclusion) we decided to use two predicates instead of one: `prove` and `fml`.

The predicate `fml` is used to specify the particular characteristics of each logical connective or quantifier:

```
fml(F,Pol,Pre1,FreeV,S,F1,F2,FUnE,FUnE1,FrV,PrV,Lim1,Lim2,V,PrN,Cp,Cp1)
```

succeeds if `F` is a first-order formula but not atomic. The parameter `Pol` is its polarity (either 0 or 1), `Pre1` its prefix, `FreeV` a list of its "free" quantifier- and prefix-variables and `S` its unique position in the formula tree. The parameters `FrV` and `PrV` are lists of free quantifier-variables and prefix-variables of the current branch, respectively.

The parameters `F1`, `F2`, `FUnE` and `FUnE1` are of the form `(Formula,Polarity)` and are bound to Prolog terms as follows: `F1` is the first (or only) subformula of `F` (possibly later bound to a copy of the subformula), `F2` is the second subformula of `F` if `F` is a β-formula (otherwise `[]`), `FUnE` is the second subformula of `F` if `F` is an α-formula (otherwise `[]`), `FUnE1` is bound to the formula `F` (and its polarity) itself if `F` is a γ-formula. If `F` is a γ-formula `Lim1` is bound to `FrV` and `V` to the variable (strictly speaking to a copy of it later on) which is quantified in `F`. If the position of `F` belongs to a prefix-variable `Lim2` is bound to `PrV`. If necessary `PrN` will be bound to a new prefix-character of `F`. The parameter `Cp` contains a term which has to be copied later on and bound to the parameter `Cp1`. The following 13 clauses define the corresponding characteristics of the intuitionistic connectives and quantifiers:

```
fml((A,B),  1, _,_,_,(A,1),[],(B,1),[],        _,_,  [],[],[],[],  [],[]).
fml((A,B),  0, _,_,_,(A,0),(B,0),[],[],        _,_,  [],[],[],[],  [],[]).
fml((A;B),  1, _,_,_,(A,1),(B,1),[],[],        _,_,  [],[],[],[],  [],[]).
fml((A;B),  0, _,_,_,(A,0),[],(B,0),[],        _,_,  [],[],[],[],  [],[]).
fml((A<=>B),Pl,_,_,_,(((A=>B),(B=>A)),Pl),[],[],[],_,_,[],[],[],[],[],[]).
fml((A=>B), 1,_,_,_, (C,0),(D,1),[],((A=>B),1),_,PrV,[],PrV,[],_,A:B,C:D).
fml((A=>B), 0,_,FV,S,(B,0),[],(A,1),[],        _,_,  [],[],[],S^FV,[],[]).
```

[3] In intuitionistic logic we have, e.g., $\neg\neg A \not\equiv A$ and $A \Rightarrow B \not\equiv \neg A \vee B$.
[4] But notice that we *prove* a formula and do not *refute* its negation.

```
fml((~A),   1,_,_,_, (C,0),[],[],((~A),1),     _,PrV,[],PrV,[],_,   A,C ).
fml((~A),   0,_,FV,S,(A,1),[],[],[],           _,_,  [],[],[],S^FV,[],[]).
fml(all X:A,1,_,_,_, (C,1),[],[],(all X:A,1),FrV,PrV,FrV,PrV,Y,_,X:A,Y:C).
fml(all X:A,0,Pr,FV,S,(C,0),[],[],[],_,_,[],[],[],S^FV,(X,A),(S^[]^Pr,C)).
fml(ex X:A, 1,Pr,FV,S,(C,1),[],[],[],_,_,[],[],[],[],  (X,A),(S^FV^Pr,C)).
fml(ex X:A, 0,_,_,_, (C,0),[],[],(ex X:A,0), FrV,_,  FrV,[],Y,[],X:A,Y:C).
```

We use a similar technique for Skolemization as leanTAP. That is we replace the quantified variable by the Skolem-term `S^FV^Pr` in the current formula `F` where `S` is the position of `F` (in the formula tree), `FV` its free quantifier- and prefix-variables and `Pr` its prefix (which we need later on). The prefix-constants have a similar format, namely `S^FV`.[5]

The predicate actually performing the proof search is

```
prove([(F,Pol),Pre,FreeV,S],UnExp,Lits,FrV,PrV,VarLim,[PU,AC])
```

It succeeds if there is a (classical) closed tableau for `F`. The parameters `UnExp` and `Lits` represent lists of formulae not yet expanded and the atomic formulae on the current branch, respectively. The parameter `VarLim` is a positive integer used to initiate backtracking (in order to obtain completeness within Prolog's depth-first search). In case of success, `PU` is bound to the prefixes of the atomic formulae which have closed the tableau, i.e. it contains pairs of prefixes `Pre1=Pre2`. The parameter `AC` is bound to a list containing the free variables (which might be bound to Prolog terms) of the proven formula and the corresponding prefixes, i.e. it contains pairs of `[Variable,Prefix]`. The other parameters have been explained before.

```
prove([(F,Pol),Pre,FreeV,S],UnExp,Lits,FrV,PrV,VarLim,[PU,AC]) :-
  fml(F,Pol,Pre1,FreeV,S,F1,F2,FUnE,FUnE1,FrV,PrV,Lim1,Lim2,V,PrN,Cp,Cp1),
  !, \+length(Lim1,VarLim), \+length(Lim2,VarLim),
  copy_term((Cp,FreeV),(Cp1,FreeV)),  append(Pre,[PrN],Pre1),
  (FUnE =[] -> UnEx2=UnExp  ; UnEx2=[[FUnE,Pre1,FreeV,r(S)]|UnExp]),
  (FUnE1=[] -> UnExp1=UnEx2 ; append(UnEx2,[[FUnE1,Pre,FreeV,S]],UnExp1)),
  (var(V)   -> FV2=[V|FreeV], FrV1=[V|FrV], AC2=[[V,Pre1]|AC1] ;
               FV2=FreeV,     FrV1=FrV,     AC2=AC1),
  (var(PrN) -> FreeV1=[PrN|FV2], PrV1=[PrN|PrV] ; FreeV1=FV2, PrV1=PrV),
  prove([F1,Pre1,FreeV1,l(S)],UnExp1,Lits,FrV1,PrV1,VarLim,[PU1,AC1]),
  (F2=[] -> PU=PU1, AC=AC2 ;
  prove([F2,Pre1,FreeV1,r(S)],UnExp1,Lits,FrV1,PrV1,VarLim,[PU2,AC3]),
  append(PU1,PU2,PU), append(AC2,AC3,AC)).
```

It depends on the actual formula `F` which steps are performed to expand `F`. After checking whether the depth-bound `VarLim` is reached, we make a copy of the specified Prolog term.[6] This is used to make a copy of those formulae which have to be kept among the unexpanded formulae or to insert a Skolem-term into a formula (in each case the free variables `FreeV` are not renamed). The current prefix is extended and the list of formulae not yet expanded is extended accordingly. We also have to add the quantifier- or prefix-variable to the corresponding lists, before expanding the subformulae.

[5] This "liberalized" Skolemization is in fact correct and complete. Just consider this Skolemization as an technique to check if the *reduction ordering* is acyclic and consider appropriate copies of the corresponding subformulae.

[6] `Lim1`/`Lim2` are used to restrict the first-order/intuitionistic *multiplicity*.

```
prove([(Lit,Pol),Pre|_],_,[[(L,P),Pr|_]|Lits],_,_,_,[PU,AC]) :-
   ( Lit=L, Pol is 1-P, (Pol=1 -> PU=[Pre=Pr] ; PU=[Pr=Pre]), AC=[]) ;
   prove([(Lit,Pol),Pre|_],[],Lits,_,_,_,[PU,AC]).
prove(Lit,[Next|UnExp],Lits,FrV,PrV,VarLim,PU_AC) :-
   prove(Next,UnExp,[Lit|Lits],FrV,PrV,VarLim,PU_AC).
```

The last two clauses remain almost unchanged from lean*TAP*. The first clause closes the current branch if the two atomic formulae `Lit` and `L` (with different polarity) can be unified.[7] The list `PU` of prefixes to be unified is extended accordingly. The last clause adds `Lit` to the list of atomic formulae on the current branch and selects another formula from those not yet expanded.

2.2 T-String Unification

After finding a closed tableau we have to unify the prefixes of those atomic formulae which have closed the tableau. These prefixes are stored in the list `PU`. A *prefix* of an atomic formula is a string and essentially describes the position of this formula in the formula tree (see [10]). Furthermore an *additional condition* on all (universal) variables has to be checked.[8] These variables are stored in the list `AC`.

The predicate `t_string_unify(PU,AC)` succeeds if all prefixes in `PU` can be unified and the additional condition for all variables in `AC` holds. In this case an intuitionistic proof can be obtained from the (classical) proof found in the first step. Otherwise we have to look for an other (classical) proof.

The two prefixes `S` and `T` are unified using the predicate `tunify(S,[],T)`.[9] A description of this predicate together with all the theoretical details are explicitly explained in [6].

```
t_string_unify([],AC)      :- addco(AC,[],final).
t_string_unify([S=T|G],AC) :- flatten([S,_],S1,[]), flatten(T,T1,[]),
                              tunify(S1,[],T1), addco(AC,[],t),
                              t_string_unify(G,AC).

tunify([],[],[]).
tunify([],[],[X|T])        :- tunify([X|T],[],[]).
tunify([X1|S],[],[X2|T])   :- (var(X1) -> (var(X2), X1==X2);
                              (\+var(X2), X1=X2)), !, tunify(S,[],T).
tunify([C|S],[],[V|T])     :- \+var(C), !, var(V), tunify([V|T],[],[C|S]).
tunify([V|S],Z,[])            :- V=Z, tunify(S,[],[]).
tunify([V|S],[],[C1|T])       :- \+var(C1), V=[], tunify(S,[],[C1|T]).
tunify([V|S],Z,[C1,C2|T])     :- \+var(C1), \+var(C2), append(Z,[C1],V),
                                 tunify(S,[],[C2|T]).
tunify([V,X|S],[],[V1|T])     :- var(V1), tunify([V1|T],[V],[X|S]).
tunify([V,X|S],[Z1|Z],[V1|T]) :- var(V1), append([Z1|Z],[Vnew],V),
                                 tunify([V1|T],[Vnew],[X|S]).
tunify([V|S],Z,[X|T])         :- (S=[]; T\=[]; \+var(X)) ->
                                 append(Z,[X],Z1), tunify([V|S],Z1,T).
```

[7] Note that for this purpose we have to use *sound* unification (i.e. with occurs check).

[8] Let σ_Q and σ_J be the term-/prefix-substitution. For all free variables u and all Skolemized variables v occurring in $\sigma_Q(u)$ the condition $|\sigma_J(\text{prefix}(u))| \geq |\sigma_J(\text{prefix}(v))|$ must hold.

[9] The variables representing the prefixes may be instantiated with nested lists. Therefore the predicate `flatten` is necessary which can be implemented as follows:

```
flatten(A,[A|B],B) :- (var(A); A=_^_), !.   flatten([],A,A).
flatten([A|B],C,D) :- flatten(A,C,E), flatten(B,E,D).
```

The predicate `addco` is used to check the interaction condition between the term- and prefix-substitution mentioned above.

```
addco(X,_,_)              :- (var(X); X=[[]]), !.
addco([[X,Pre]|L],[],Ki) :- !, addco(X,Pre,Ki), addco(L,[],Ki).
addco(_^_^Pre1,Pre,Ki)   :- !, ( Ki=final -> flatten(Pre1,S,[]),
                                 flatten(Pre,T,[]), append(S,_,T) ;
                                 \+ \+ t_string_unify([Pre1=Pre],[]) ).
addco(T,Pre,Ki)     :- T=..[_,T1|T2],!, addco(T1,Pre,Ki), addco(T2,Pre,Ki).
addco(_,_,_).
```

The following goal succeeds if `F` can be proven *intuitionistically* without using more than `VarLim` (quantifier- and prefix-) variables on each branch:

```
prove([(F,0),[],[],1],[],[],[],[],VarLim,[PU,AC]), t_string_unify(PU,AC).
```

3 Performance

Although the presented implementation is comparably short its performance seems to be quite good. We will show some experimental results comparing ileanTAP with the tableau prover ft [8]. We also provide the timing of lean*TAP* to point out the correlation between lean*TAP* and ileanTAP. All three provers perform an iterative deepening. The following problems are taken from [8] and measured on a Sun SPARC10 (times are given in seconds; "–" means that no proof was found within 150 seconds).

No.	ft	ilean TAP	lean TAP
1.1	0.03	0.07	0.02
1.2	0.56	0.12	0.05
1.3	–	0.35	0.08
1.4	0.01	0.05	< 0.01
1.5	1.81	0.13	0.02
1.6	18.67	0.20	0.07
1.7	0.16	0.03	0.02
1.8	–	0.13	0.08
2.1	1.73	–	–
2.2	3.61	–	–
2.3	6.08	–	–
3.1	< 0.01	0.02	< 0.01
3.2	0.07	1.72	< 0.01
3.3	0.01	0.05	0.02
3.4	< 0.01	0.17	0.02
3.5	2.66	–	0.65
4.1	10.53	–	2.40
4.2	11.78	–	4.30
5.1	< 0.01	0.08	< 0.01
5.2	0.03	3.17	0.30
5.3	0.76	121.98	8.05
6.1	< 0.01	< 0.01	< 0.01
6.2	0.01	0.48	0.60
6.3	0.01	0.10	< 0.01
6.4	< 0.01	0.08	0.02
6.5	< 0.01	0.15	< 0.01
6.6	< 0.01	0.02	< 0.01
6.7	< 0.01	0.02	< 0.01
6.8	0.01	0.02	0.02
6.9	0.01	0.03	0.03
6.10	< 0.01	0.10	0.03
6.11	< 0.01	0.03	< 0.01
6.12	0.04	–	0.02
6.13	0.07	4.26	0.07
6.14	0.01	0.10	0.03
6.15	0.04	–	0.15
7.1	< 0.01	0.02	< 0.01
7.2	0.05	0.48	< 0.01
7.3	3.98	32.85	0.03

Table 1. Performance of ileanTAP with eclipse Prolog Version 3.5.2

ileanTAP yields very good results on the problems of group 1 ("alternations of quantifiers"), 3 ("Pelletier's problems 39–43") and 6 ("simple"). It has a similar performance on the problems of group 5 ("unify") and 7 ("problematic"), but behaves very poorly on problems of group 2 ("append") and 4 ("existence"). The problems of group 2 are even too hard for the classical prover lean*TAP*.

Altogether, a remarkable result keeping in mind that ft consists of about 200 kbytes C-source code (whereas ileanTAP has a size of about 4 kbytes).[10]

[10] The prover in [9] is a bit slower than ft on the simpler problems. It cannot solve problem no. 5.3 but prove all other problems in less than one second.

4 Conclusion

We presented a Prolog implementation for a first-order intuitionistic theorem prover. We encode the additional non-permutabilities arising in intuitionistic logic in a sophisticated way namely by an additional string unification which yields a quite good performance. Due to the compact code the program can easily be modified for special purposes or applications.

Since the particular characteristics of the logical connectives and quantifiers are specified in a separate way, it is easy to adapt the prover to other non-classical logics. Replacing the last 8 clauses of the predicate `fml` by the following clauses

```
fml((A=>B), 1,_,_,_, (A,0),(B,1),[],[],        _,_,  [],[],[],[], [],[]).
fml((A=>B), 0,_,_,_, (B,0),[],(A,1),[],        _,_,  [],[],[],[], [],[]).
fml((~A),   1,_,_,_, (A,0),[],   [],[],        _,_,  [],[],[],[], [],[]).
fml((~A),   0,_,_,_, (A,1),[],   [],[],        _,_,  [],[],[],[], [],[]).
fml(all X:A,1,_,_,_, (C,1),[],[],(all X:A,1),FrV,_, FrV,[],Y,[],X:A,Y:C).
fml(all X:A,0,Pr,FV,S,(C,0),[],[],[],_,_, [],[],[],[],(X,A),(S^FV^Pr,C)).
fml(ex X:A, 1,Pr,FV,S,(C,1),[],[],[],_,_, [],[],[],[],(X,A),(S^FV^Pr,C)).
fml(ex X:A, 0,_,_,_, (C,0),[],[],(ex X:A,0), FrV,_, FrV,[],Y,[],X:A,Y:C).
```

and adding the following 4 clauses

```
fml([](A),  1,_,_,_, (C,1),[],[],([](A),1),_,PrV,[], PrV,[],_,     A,C ).
fml([](A),  0,_,FV,S,(A,0),[],[],[],       _,_,  [], [], [],S^FV, [],[]).
fml(<>(A),  1,_,FV,S,(A,1),[],[],[],       _,_,  [], [], [],S^FV, [],[]).
fml(<>(A),  0,_,_,_, (C,0),[],[],(<>(A),0),_,PrV,[], PrV,[],_,     A,C ).
```

immediately yields a program implementing a theorem prover for the first-order modal logic S4 (where "`[]`" and "`<>`" represent the corresponding modal operators). Modifying the algorithm for T-string unification accordingly leads also to provers for the modal logics D, D4, S5, and T (see [6, 7] for details).

Of course, there is still room for further research. The current implementation is not a decision procedure for the propositional intuitionistic logic (which is decidable), since we need *multiplicities* already in this fragment. For example it would be interesting to integrate some techniques from [3] to get a decision procedure for the propositional fragment of intuitionistic logic.

References

1. B. Beckert and J. Posegga. leanTAP: Lean Tableau-Based Theorem Proving. *Proc. CADE-12*, LNAI 814, pp. 793–797, Springer Verlag, 1994.
2. B. Beckert and J. Posegga. leanTAP: Lean Tableau-based Deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
3. R. Dyckhoff. Contraction-free Sequent Calculi for Intuitionistic Logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.
4. M. C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer Verlag, 1990.
5. J. Otten, C. Kreitz. A Connection-Based Proof Method for Intuitionistic Logic. *Proc. 4^{th} TABLEAUX Workshop*, LNAI 918, pp. 122–137, 1995.
6. J. Otten, C. Kreitz. T-String-Unification: Unifying Prefixes in Non-Classical Proof Methods. *Proc. 5^{th} TABLEAUX Workshop*, LNAI 1071, pp. 244–260, 1996.
7. J. Otten, C. Kreitz. A Uniform Proof Procedure for Classical and Non-Classical Logics. *KI-96: Advances in Artificial Intelligence*, LNAI, Springer Verlag, 1996.
8. D. Sahlin, T. Franzen, S. Haridi. An Intuitionistic Predicate Logic Theorem Prover. *Journal of Logic and Computation*, 2(5):619–656, 1992.
9. T. Tammet. A Resolution Theorem Prover for Intuitionistic Logic. *Proc. CADE-13*, LNAI, Springer Verlag, 1996.
10. L. Wallen. *Automated deduction in nonclassical logic.* MIT Press, 1990.