λleanTAP: lean deduction in λ Prolog

Alexei Lisitsa

Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, U.K.
A.Lisitsa@csc.liv.ac.uk

Abstract. In this note, we describe λleanTAP, lean tableaux prover for first-order logic, implemented in the higher-order logic programming language λProlog. It is derivative of well-known leanT^AP prover [5, 6], but makes use of the higher-order unification and λ-terms as data structures in λ*Prolog*. This allows us to re-implement leanT^AP in more direct and declarative way, without using the "non-logical" *copy_term* predicate and with λ-terms representing the quantifier's bindings.
Overall the efficiency of such λleanTAP proves to be comparable with the original leanT^AP. Finally, we show the experimental results for a subset of Pelletier's problems [17].

1 Introduction

In [3, 5, 6], Becker and Posegga presented very elegant and efficient first-order tableaux-based theorem prover, called leanT^AP. It makes direct and essential use of the search mechanism provided by Prolog, which allows leanT^AP to be made so small - it consists of only five clauses - and very efficient at the same time. Based on that, Becker and Possega have formulated the *lean deduction* principle for automated theorem proving, that is *to achieve maximal efficiency from minimal means* and have demonstrated that logic programming languages are very suitable for this purpose. Indeed, if we already have very efficient and optimized search mechanisms in Prolog implementations, why not use these as much as possible in theorem provers?

Since, the introduction of leanT^AP , there have been various developments in lean deduction, including, among others,

- implementations of lean provers for different calculi/logics [11, 16];
- proof theoretical investigation of the basic principles of leanT^AP [11];
- lean induction for tableaux calculi [1];
- lean provers in non-classical logic programming languages [12].

This note reports on the development which falls into the last category: using the non-classical logic programming language λProlog for re-implementing leanT^AP. λProlog is a higher-order strongly typed logic programming language [13, 14]. Main features which distinguish λProlog from standard Prolog are λ-terms as data structures and higher-order unification. This makes λProlog very suitable for programs which manipulate syntactical structures involving the notions of abstractions, scope, bound and free variables, substitution instances, etc. Applications of λProlog in automated theorem proving are disscussed e.g. in [9, 10]. Notice that, unlike the case of leanT^AP, practical efficiency of theorem provers was not the main concern in these papers. The implementation of λleanTAP has shown that lean deduction can be carried out succintctly in λProlog as well.

Our reasons in choosing λProlog for re-implementing leanT^AP were pure pragmatical in the beginning: We had been working with the λ*Clam* proof planning system[18], implemented in λProlog, and needed a simple, but efficient, first-order theorem prover to be used as a pat of first-order temporal theorem proving within λ*Clam*. However, it turned out that the re-implementation is not as straightforward as expected, and it is interesting in its own right, making use of powerful features of λProlog.

There are several different implementations of λProlog [13]. Because λ*Clam* is implemented in Teyjus λProlog[20, 15], it was a natural choice for λleanTAP. Although it has not been checked, we believe that with little effort it can be ported to any other implementation of λProlog. Thus, in this note we assume Teyjus syntax.

In the original leanT^AP object variables in first-order formulae are represented directly by Prolog variables. Because of that, special care must be taken to avoid clashes between variables and to generate fresh variables, needed in the tableax construction. So, leanT^AP uses the *copy_term* predicate of standard Prolog to supply fresh (non-bound) copies of variables. Although *copy_term* is a part of the ISO standard of Prolog, it does not have clear "logical" semantics and it is not a part of pure logic programming paradigm. For example, λProlog lacks *copy_term* altogether and there is no easy way to re-introduce it. However, λProlog provides an alternative and more direct way of dealing with variables and their bindings, that is by using λ-terms.

In the next section the implementation of λleanTAP in comparison with the original leanT^AP is disscussed. Then, in Section 3 performance of λleanTAP is disscussed. Finally, Section 4 outlines possible future directions.

2 From leanT^AP to λleanTAP

2.1 leanT^AP

The source of original leanT^AP [4] is shown in Figure 1. For the detailed description of how it works see [3, 4]. Here we restrict ourselves just to the brief outline from [4]:

> The predicate `prove(Fml,UnExp,Lits,Freev,VarLim)` 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 tableaux; the parameters `Fml`, `UnExp`, and `Lits` represent the current branch: `Fml` is the formula being expanded, `UnExp` holds a list of formulae not yet expanded, and `Lits` is a list of the literals present ob the current branch. `FreeV` is alist 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`.
>
> leanT^AP uses Prolog syntax for first-order fromulae: 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 $p(0) \wedge (\forall n(\neg p(n) \vee p(s(n)))))$.

```
% 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).
```

Fig. 1. leanT^AP source code

As it is, leanT^AP works with closed first-order formulae in Skolemized negation normal form, but it can be easily extended to full first-order logic by adding appropriate tableaux rules. Alternatively, a simple translator to the Skolemized negation normal form can be used at the pre-processing stage.

Everything from the above brief outline of basic principles, except formulae syntax, holds also for λleanTAP.

2.2 Syntax of formulae in λleanTAP

Modulo non-essential differences, the *quantifier-free* part of the syntax for the first-order formulae representation in λleanTAP is the same as in leanT^AP. The case of quantifiers, however, is treated differently, because of the explicit representation of the binding structure via λ-terms. Figure 2 shows the formula syntax for both implementations.

Formulae	leanT^AP	λleanTAP
$p \wedge q$	(p,q)	(p and q)
$p \vee q$	(p;q)	(p or q)
$\neg p$	-p	neg(p)
$\forall x p(x)$	all(X,p(X))	(forall x\ p(x))
$p \rightarrow q$	(p => q)	(p imp q)
$p \leftrightarrow q$	(p <=> q)	(p eqv q)
$\exists x p(x)$	ex(X,p(X))	(exists x\ p(x))

Fig. 2. Formula syntax

Notice that only the variants above the double line in Figure 2 are used in the negated normal form. Those under the double line are needed if full first-order syntax is allowed in the input for the theorem prover.

Let us take closer look at the teratment of quantifiers via λ-terms. Consider the formula $\forall x\ p(x)$, which is represented in λleanTAP by the term `(forall x\ p(x))`, which, in turn, is λProlog's syntactic representation of the λ-term (forall λx (p x)). Thus, $x\backslash$ stands here for the lambda-abstracted variable and brackets stands for the application operation[1]. Both `forall` and `p` are considered as (higher-order) constants.

Such an higher-order abstract syntax representation and the way λProlog deals with λ-terms has many advantages[14] for the representation of formulae:

- the binding structure of quantifiers is represented explicitly;
- the binding structure may be analysed by using higher-order unification;
- the terms considered modulo α-conversion (that means, e.g. λ-terms presenting $\forall x\ \varphi(x)$ and $\forall y\ \varphi(y)$ are the *same*);
- the instantiation of the quantifier in a formula by a term is implemented by the β-conversion rule, and all necessary renaming operations are done automatically.

As a result, the manipulation of binding structures in λProlog becomes very simple and transparent. Consider the following two clauses defining a predicate `strip` from λleanTAP :

```
strip (forall A) (A X).
strip (exists A) (A (A ex)).
```

The first line implements instantiation of an universal quantifier in a formula by a *new* free variable. The second line implements the Skolemization of a formula by the formula itself [4, 2]. That is, in fact, all we need for quantifier manipulation in λleanTAP and in translation to the Skolemized negated form.

```
% Qunatifier's manipulation

strip (forall A) (A X).
strip (exists A) (A (A ex)).

%  Unification with a member of list

memberunify Element [Head|Tail] :- !,
  Element = Head;memberunify Element Tail.

% Conjunction

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

% Disjunction

prove (A or B) UnExp Lits FreeV VarLim :- !,
prove A UnExp Lits FreeV VarLim ,
prove B UnExp Lits FreeV VarLim.

%  Universal quantification

prove (forall Fml) UnExp Lits FreeV VarLim :- !,
length FreeV L, L < VarLim ,
       strip (forall Fml) (Fml X),
       append UnExp [(forall Fml)] UnExp1,
       prove (Fml X) UnExp1 Lits [X|FreeV] VarLim.

% Closing branches:

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


% Extending branches

prove Lit [Next|UnExp] Lits FreeV VarLim  :-
    prove Next UnExp [Lit|Lits] FreeV VarLim.
```

Fig. 3. λleanTAP source code

2.3 λleanTAP

The source code for λleanTAP is shown in Figure 3.

Notice that leanT^AP requires sound unification with the occurs check (see predicate `unify` at line 13 in leanT^AP code), so one needs, depending on particular implementation of Prolog, either to implement such sound unification, or to switch on such an option. In λProlog sound unification is as the standard build-in procedure. This explains the absence of any `unify` predicate at lines 15–17 of λleanTAP code (equality is used instead). The predicate `memberunify` appears at the line 17 (and its definition at lines 3–4) in order to clarify the "most obscure"[4] part of leanT^AP code, that is the lines 12–13. Here we follow the suggestion of the authors of leanT^AP in [4].

λProlog is a strongly typed language and all defined (higher-order) constants need to be typed. The type system in λProlog is very flexible and allows type variables in type declarations to provide polymorphic typing. The following signature file contains type declarations for all predicates of λleanTAP with A being a type variable and `o` denoting a primitive type. Of course, more concrete typing might be used instead.

```
sig lambdaleanTAP.

type append list               A -> list A -> list A -> o.
type length list               A -> int -> o.
type memberunify               A -> list A -> o.
type strip                     A -> A -> o.
type prove                     A -> (list A) -> (list A) ->
                                      (list  A) -> int -> o.
type and,or,imp                A -> A -> A.
type eqv                       A -> A -> A.
type neg                       A -> A.
type forall                    (A -> A) -> A.
type exists                    (A -> A) -> A.
type false                     A.
type ex                        A.
```

Fig. 4. λleanTAP's signature file with type information

3 λLeanTap and its perfomance

Figure 5 shows experimental results for a subset[2] of Pelletier's problems used for testing first-order theorem provers [17]. We compare the results for λleanTAP running on Teyjus Prolog, version 1.0-b33 with the results for leanT^AP running on Sicstus Prolog,

[1] λProlog allows both (XY) and $X(Y)$ to denote application of X to Y.

[2] This is the same subset as in [5]

version 3.9.1 and on SWI-Prolog, version 5.0.7. For the experiments an Intel system (700Mhz, 128Mb RAM) running RedHat 7.3 Linux was used. The results presents average time (in milliseconds) for ten attempts. "0 msec" means "average time is less than 0.5 msec" or "not measurable". The notation "-" (for λleanTAP and Pelletier 34) means "did not return in 1 hour".

Pelletier's No.	leanT^AP (Sicstus)	leanT^AP(SWI-Prolog)	λleanTAP (Teyjus)
17	0	0	0
18	0	0	0
19	1	0	1
20	0	0	0
21	1	1	0
22	0	1	0
23	0	0	0
24	0	1	2
25	0	1	0
26	0	1	1
27	0	0	0
28	0	0	0
29	1	1	2
30	0	0	0
31	0	0	1
32	0	1	1
33	0	0	1
34	43	106	-
35	0	0	0
36	0	0	1
37	1	2	11
38	8	28	55
39	0	0	0
40	1	0	1
41	1	0	1
42	0	3	2
43	4	15	10
44	0	3	1
45	2	3	4
46	150	365	2

Fig. 5. λleanTAPś vs leanT^APś performance for Pelletier's problems

The figure suggests that the behaviour of λleanTAP should be understood better. The cases where the results differ substantially, such as Pelletier's problems 34,37,38 and 46 are of special interest. Notice, that "did not return" result for Pelletier 34 was also reported for leanT^AP and Sicstus 2.1 in the first publication on leanT^AP [3]. We plan to compare the performance of these provers (on different implementations of Prolog and λProlog) on the bigger set of problems from TPTP Problem Library[19].

4 Conclusions

We have re-implemented leanT^AP in λProlog and have shown it has comparative perfomance to the original leanT^AP . More importantly, the implementation is cleaner using essentially declarative features.

In λLeanTap the facilities of λProlog have been used to modify leanT^AP in a rather conservative way and only on the level of formula manipulation. We believe, however, that higher-order features of λProlog may be used in lean tableaux provers in more radical ways, especially in the area of non-classical modal and temporal logic for guiding proof search, loop conditions checking, etc [7]. This is the subject of future work. Also, possibility of dealing with equality via higher-order unification will be explored.

5 Acknowledgements

I am grateful to Boris Konev for drawing my attention to the exciting area of lean deduction, and to him, Anatoli Degtyarev and Michael Fisher, for the discussions of various aspects of this work.

References

[1] Matthias Baaz, Uwe Egly, Christian G. Fermller. Lean Induction Principles for Tableaux. In *Proc. of Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '97, Pont-à-Mousson, France, May 13-16, 1997*, Springer, Lecture Notes in Computer Science, vol. 1227, 1997, pp 62-75.

[2] B. Beckert, R. Hähnle and P. H. Schmitt. The Even More Liberalized δ-rule in Free Variable Semantic Tableaux. In Gottlob, G., Leitsch, A., and Mundici, D. (eds),*Proceedings of 3rd Kurt Gödel Colloquium (KGC)*, Brno, Czech Republic, LNCS 713, Springer, pp 108–119.

[3] B. Beckert and J. Posegga. leanT^AP: lean tableau-based theorem proving. In *Proc. of 12th International Conference on Automated Deduction*, pages 793–797. Springer-Verlag LNAI 814, 1994.

[4] B. Beckert and J. Posegga. The leanT^AP-FAQ: Frequently Asked Questions about leanT^AP, WWW: http://emmy.ira.uka.de/ posegga/leantap

[5] B. Beckert and J. Posegga. leanT^AP: Lean, Tableau-based Deduction. *Journal of Automated Reasoning*, Vol. 15, No. 3, pages 339-358, 1995.

[6] B. Beckert and J. Posegga. Logic programming as a basis for lean automated deduction. *The Journal of Logic Programming*, Vol. 28, No. 3, pages 231-236, 1996.

[7] J. Brotherston, A. Degtyarev, M. Fisher and A. Lisitsa, Searching for Invariants using Temporal Resolution, in Proceeding of 9th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Tbilisi, Georgia, 2002, LNAI 2514, Springer, pp 86–101

[8] A. Degtyarev and M. Fisher. Towards first-order temporal resolution. In *Proccedings of KI-2001*, volume 2174 of *LNAI*, 2001.

[9] Amy Felty and Dale Miller. Specifying Theorem Provers in a Higher-Order Logic Programming Language. In *Ninth International Conference on Automated Deduction*, Springer-Verlag LNCS, 310, May 1988.

[10] Amy Felty. Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. *Journal of Automated Reasoning*, 11(1):43-81, August 1993.

[11] M. Fitting leanT^AP Revisited. *Journal of Logic and Computation*, 8:33–47, 1998

[12] Joshua S. Hodas, Naoyuki Tamura lolliCop - A Linear Logic Implementation of a Lean Connection-Method Theorem Prover for First-Order Classical Logic. In *Proc. of IJCAR 2001*, pp 670-684

[13] λ*Prolog* home page, http://www.cse.psu.edu/ dale/lProlog/

[14] Gopalan Nadathur. The Metalanguage Lambda Prolog and Its Implementation. To appear in@ *Proceedings of the Fifth International Symposium on Functional and Logic Programming,* H. Kuchen and K. Ueda (eds), March 2001.

[15] Gopalan Nadathur and Dustin J. Mitchell. System Description: Teyjus-A Compiler and Abstract Machine Based Implementation of Lambda Prolog. In *Sixteenth Conference on Automated Deduction, H. Ganzinger (ed.)*, pages 287-291, 1999.

[16] J. Otten and W. Bibel. LeanCoP: lean connection-based theorem proving. In *Proceedings of the Third Intenational workshop on First-Order Theorem Proving*, pages 152–157. University of Koblenz, 2000.

[17] F. J. Pelletier. Seventy-Five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning,* 2, 191-216.

[18] J. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with lambdaclam. In *Proc. of CADE'15*, volume 1421 of LNAI, 1998.

[19] G. Sutcliffe and C. Suttner. The TPTP Problem Library for Automated Theorem Proving. available at http://www.cs.miami.edu/ tptp/

[20] Teyjus web page, http://teyjus.cs.umn.edu/