leanTAP: Lean Tableau-based Deduction*

Bernhard Beckert & Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe, Germany
{beckert|posegga}@ira.uka.de

Abstract.

```
"prove((E,F),A,B,C,D) :- !, prove(E,[F|A],B,C,D).
 prove((E;F),A,B,C,D) :- !, prove(E,A,B,C,D), prove(F,A,B,C,D).
 prove(all(H,I),A,B,C,D) :- !,
     \+length(C,D), copy_term((H,I,C),(G,F,C)),
     append(A,[all(H,I)],E), prove(F,E,B,[G|C],D).
 prove(A,_,[C|D],_,_) :-
     ((A= -(B); -(A)=B)) -> (unify(B,C); prove(A,[],D,_,_)).
 prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D)."
```

implements a first-order theorem prover based on free-variable semantic tableaux. It is complete, sound, and efficient.

1 Introduction

The Prolog program listed in the abstract implements a complete and sound theorem prover for first-order logic; it is based on free-variable semantic tableaux (Fitting, 1990). We call this *lean deduction*: the idea is to achieve maximal efficiency from minimal means. We will see that the above program is indeed very efficient—not *although* but *because* it is extremely short and compact.

Our approach surely does not lead to a deduction system which is superior to highly sophisticated systems like Otter (McCune, 1990) or Setheo (Letz *et al.*, 1992); these are better on solving difficult problems. However, many applications do not require deduction which is as complex as the state of the art in automated theorem proving can handle. Furthermore, there are often strong constraints on the time allowed for deduction. Our approach can be particularly useful in such areas: it offers high inference rates on simple to moderately complex problems and a high degree of adaptability.

Another important argument for lean deduction is safety: It is easily possible to verify a couple of lines of standard Prolog; verifying thousands of lines of C code, however, is hard—if not impossible—in practice.

Satchmo (Manthey & Bry, 1988) can be regarded the earliest application of lean theorem proving. The core of Satchmo is about 15 lines of Prolog code,

* To appear in the Journal of Automated Reasoning.

Unifikation in PROLOG

Explizite Unifikation

– „=“ (ohne Occur-Check)

– „`unify`“ (mit Occur-Check, langsamer)

Unterschiede zum Beispiel für

```
| ?- p(X,X) = p(Y,f(Y)).
```

und

```
| ?- unify(p(X,X),p(Y,f(Y))).
```

Implizite Unifikation (ohne Occur-Check) jeweils zwischen Unterziel und Kopf einer Regel.

Unifikation in (Abschlußregel) für Tableauzweig benötigt Occur-Check für Korrektheit.

Negation in PROLOG

Negations-Operator „`\+`“

Negation as failure:
Die leere Substitution ist Antwort für `\+goal`,
falls die Suche nach einer Antwort für `goal` fehlschlägt.

Choice points in PROLOG

Choicepoint (=Verzweigung des BSB) erzeugt, falls

- Unterziel mit mehreren Regelköpfen unifizierbar
- durch Oder-Operator „`p ; q`“

leanTAP: Das Programm

Beckert & Posegga (1995)

```
prove((A,B),UnEx,Lits,FreeV,Lim) :- !,
   prove(A,[B|UnEx],Lits,FreeV,Lim).

prove((A;B),UnEx,Lits,FreeV,Lim) :- !,
   prove(A,UnEx,Lits,FreeV,Lim),
   prove(B,UnEx,Lits,FreeV,Lim).

prove(all(X,Fml),UnEx,Lits,FreeV,Lim) :- !,
   \+ length(FreeV,Lim),
   copy_term((X,Fml,FreeV),
             (X1,Fml1,FreeV)),
   append(UnEx,[all(X,Fml)],UnEx1),
   prove(Fml1,UnEx1,Lits,[X1|FreeV],Lim).

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

prove(Lit,[Next|UnEx],Lits,FreeV,Lim) :-
   prove(Next,UnEx,[Lit|Lits],FreeV,Lim).
```

leanT^AP: Die Idee

Vorteilhafte Nutzung von PROLOG:

- Backtracking
- Indexing
- Datenstrukturen (Terme)
- Unifikation

Die Anfrage

Aktueller Zweig

```
prove(Fml,UnEx,Lits,FreeV,Lim)
```

ist für gegebene geschlossene Formel `Fml` in NNF ohne $\exists$ erfolgreich, wenn es ein geschlossenes Tableau für `Fml` mit höchstens `Lim` freien Variablen pro Zweig gibt.

Entwursentscheidungen

- inkrementelle Tiefensuche mit BT
- Vollständigkeitsmodus m, wobei in $m(i)$ alle Tableaus mit höchstens i freien Variablen pro Ast
- expliziter Zugriff nur auf den aktuellen Ast
  andere noch offene Äste und Choice points (effizient) von PROLOG verwaltet (structure sharing von Termen, Kopieren der Variablenbindungen anstatt ganzer Terme);
- Darstellung des aktuellen Astes:

`Fml`	nächste zu betrachtende Formel
`UnEx`	weitere noch zu betrachtende Formeln
`Lits`	schon betrachtete Literale

  - Separate Verwaltung von `Fml` im ersten Argument, da PROLOG-Regeln in Hashtabelle verwaltet werden mit Prädikat- und erstem Funktionssymbol des Kopfliterals als Schlüssel
  - Trennung in `UnEx` und `Lits` vermeidet wiederholte Suche nach Nichtliteral bzw. Literal
- Eingabe in NNF ohne $\exists$
  (Regeln nur für $\wedge$, $\vee$, $\forall$).

Formelrepräsentation

Konjunktion `(A,B)` Infix

Disjunktion `(A;B)` Infix

Negation `-A` Präfix

Allquantor `all(X,A)` Präfix

Beispiel.

```
(all(Y,(-(s(Y));-(q(Y)))),
 all(Z,(-(p(Z));q(Z);r(Z))),
 (p(c);q(d)),
 all(W,(-(q(W)),-(r(W));s(W))),
 all(X,(-(p(X));-(r(X))))
)
```

Eingabe von Formelmengen als Konjunktion ihrer Elemente.

Start:

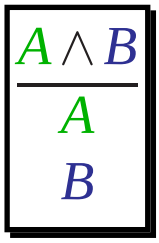
```
| ?- prove(Fml,[],[],[],Lim).
```

erfolgreich falls es geschlossenes Tableau für `Fml` in $m(\text{Lim})$ gibt.

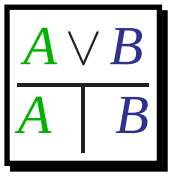
Konjunktion & Disjunktion

Aktueller Zweig

```
prove(Fml,UnEx,Lits,FreeV,Lim)
```




```
prove((A,B),UnEx,Lits,FreeV,Lim):-
  !,
  prove(A,[B|UnEx],Lits,FreeV,Lim).
```




```
prove((A;B),UnEx,Lits,FreeV,Lim):-
  !,
  prove(A,UnEx,Lits,FreeV,Lim),
  prove(B,UnEx,Lits,FreeV,Lim).
```

Diskussion

```
prove((A,B),UnEx,Lits,FreeV,Lim) :- !,
   prove(A,[B|UnEx],Lits,FreeV,Lim).

prove((A;B),UnEx,Lits,FreeV,Lim) :- !,
   prove(A,UnEx,Lits,FreeV,Lim),
   prove(B,UnEx,Lits,FreeV,Lim).
```

Die letzte erzeugte Formel ist erstes Element von UnEx; Daher: depth_first bis auf Typ γ.

Keine Choicepoints erzeugt (deterministische Regeln).

Cut bewirkt, daß Nichtliterale nicht auf Abschluß getestet werden.

In Disjunktionsregel wird linker Zweig immer zuerst geschlossen: left_first Auswahlregel.

Universeller Quantor

Aktueller Zweig

```
prove(Fml,UnEx,Lits,FreeV,Lim)
```

$$\frac{\forall X \; Fml(X)}{Fml(X)}$$

```
prove(all(X,Fml),UnEx,Lits,FreeV,Lim):-
 !,
 \+length(FreeV,Lim),
 copy_term((X,Fml,FreeV),
           (X1,Fml1,FreeV)),
 append(UnEx,[all(X,Fml)],UnEx1),
 prove(Fml1,UnEx1,Lits,[X1|FreeV],Lim).
```

Diskussion

```
prove(all(X,Fml),UnEx,Lits,FreeV,Lim) :- !,
   \+ length(FreeV,Lim),
   copy_term((X,Fml,FreeV),
             (X1,Fml1,FreeV)),
   append(UnEx,[all(X,Fml)],UnEx1),
   prove(Fml1,UnEx1,Lits,[X1|FreeV],Lim).
```

Falls Liste der freien Variablen im Zweig `FreeV` Länge `Lim` erreicht hat, wird fail erzeugt.

Aufruf von `copy_term`

- ersetzt *genau* `X` in `Fml` durch neue Variable `X1` in `Fml1` und
- führt eine gebundene Umbenennung aus

`FreeV` verhindert, daß andere freie Variablen in `Fml` durch neue Variablen ersetzt werden (Hauptgrund für Vorhandensein von `FreeV`).

`all(X,Fml)` kommt an das *Ende* von `UnEx`, was zu *fairer* Implementierung von `select_formula` führt.

Fairneß notwendig, da `select_formula` deterministisch.

Abschluß und Erweiterung von Zweigen

Aktueller Zweig

```
prove(Fml,UnEx,Lits,FreeV,Lim)
```

$$\frac{\begin{array}{c}\text{Lit}\\ \vdots\\ \text{L}\end{array}}{\sigma}$$

σ mgu von Neg $=$ $\overline{\text{Lit}}$ und L

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

Erweiterung:

```
prove(Lit,[Next|UnEx],Lits,FreeV,Lim):-
 prove(Next,UnEx,[Lit|Lits],FreeV,Lim).
```

Diskussion

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

Abschlußregel ist einzig nichtdeterministische.

Sie ruft sich rekursiv auf, bis komplementäres Paar gefunden.

„`[]`“ verhindert, daß bei dieser Rekursion die fünfte Regel angesteuert wird.

Prolog-Implementierung von if-then-else beinhaltet Cut, der den Fall `--p = -p` verhindert (würde für `Lit = -p` durch Backtracking auftreten).

Bedingung von if-then-else erzeugt Komplement `Neg` von `Lit`.

Existenz eines mgu von `Neg` und `L` durch Unifikation *mit occur-check* geprüft.

Für Berechnung des Komplements reicht „=“, da `Neg` stets neue Variable.

Diskussion

```
prove(Lit,[Next|UnEx],Lits,FreeV,Lim):-
 prove(Next,UnEx,[Lit|Lits],FreeV,Lim).
```

Die Regel realisiert Indeterminismus von `select_mode`.

Wird aktiviert, wenn vorher mögliche Abschlüsse zu Fehlschlag führten.

Aktuelle Formel muß Literal sein, das zu `Lits` kommt.

Erstes Element von `UnEx` wird zur neuen aktuellen Formel.

`select_formula` ist deterministisch: zuletzt erzeugte Formel wird zuerst expandiert (bis auf γ-Formeln).

Zusammenfassung

leanT^AP hat Choicepoints in `select_pair` und `select_mode`.

`select_branch` (left_first) und `select_formula` deterministisch.

Anordnung der γ-Formeln als Schlange ergibt faire Implementierung von `select_formula`.

Um unerfüllbare Formel zu widerlegen, wird leanT^AP sukzessive mit `Lim = 0,1,2,...` aufgerufen: iterative gebundene Tiefensuche.

leanT^AP lauffähig mit jedem DEC-10 PROLOG (z.B.: Quintus, SICStus, Eclipse), evtl. Bibliotheken für `append`, `length`, `unify`, `copy_term` laden.

Beispiel Mengenlehre: ca. 5 msec mit `Lim = 6` auf Sun Sparc 10/20 mit Sicstus PROLOG.

`http://i12www.ira.uka.de/~posegga/leantap/leantap.html`