

How to Build an Automated Theorem Prover

Part 3a: A Tableau Prover

Jens Otten

University of Oslo



Negation Normal Form

- ▶ a formula is in **negation normal form (NNF)** if it only contains \wedge , \vee , \forall , \exists , and \neg only occurs (directly) in front of atomic formulae
- ▶ every formula F can be **translated** into a (classically) equivalent (model-preserving) formula F' that is in **negation normal form**
- ▶ **translation** of a formula into negation normal form:
 1. Eliminate all logical operators except \neg , \wedge , \vee :
 $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$, $A \rightarrow B \equiv \neg A \vee B$
 2. Push negations inward using De Morgan's laws:
 $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$, $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$,
 $\neg \forall x A \equiv \exists x \neg A$, $\neg \exists x A \equiv \forall x \neg A$
 3. Eliminate double negation: $\neg \neg A \equiv A$



Skolemization for NNF

- ▶ a formula is in **skolemized negation normal form**, if all its Eigenvariables have been replaced by **Skolem terms**.
- ▶ let F be a formula in NNF, $\forall x G$ be a subformula in F , and $\exists y_1, \dots, \exists y_n$ be the existential quantifiers "in front" of G
- ▶ then F is valid iff $F[x \setminus f^*(y_1, \dots, y_n)]$ is valid, where the Eigenvariable x (in G) is replaced by the skolem term $f^*(y_1, \dots, y_n)$ for a new function symbol f^*

Example: $\forall x \exists y p(x, y) \rightarrow \exists y \forall x p(x, y)$

- ▶ formula in NNF: $\exists x \forall y \neg p(x, y) \vee \exists y \forall x p(x, y)$
- ▶ skolemized NNF: $\exists x \neg p(x, f_1(x)) \vee \exists y p(f_2(y), y)$



Block Tableau Calculus for NNF with Free Variables

The **tableau calculus** for formulae in negation normal form (NNF) consists of one axiom and an α -rule, a β -rule, and a γ -rule.

- ▶ **axiom**

$$\frac{P, \neg P', \Delta}{\text{axiom}} \quad (\text{with } P/P' \text{ unifiable, i.e., } \sigma(P) = \sigma(P'))$$

- ▶ **α -rule**

$$\frac{A, B, \Delta}{A \vee B, \Delta} \vee$$

- ▶ **β -rule**

$$\frac{A, \Delta \quad B, \Delta}{A \wedge B, \Delta} \wedge$$

- ▶ **γ -rule**

$$\frac{A[x \setminus x^*], \exists x A, \Delta}{\exists x A, \Delta} \exists \quad (\text{new } x^*)$$

- ▶ similar to **one-sided** sequent calculus with skolemized NNF
- ▶ all rules are now **invertible**; P, P' are atomic formulae
- ▶ all literals in axioms have to unify under a **single** substitution σ

F is **valid** iff there is a **proof** for F in the block tableau calculus.



Block Tableau Calculus with Ordered Sets

- ▶ if an **ordered set** Δ is considered, the rule applications can be restricted to its **first element**; this simplifies the proof search
- ▶ the words of the calculus are of the form $F, \Delta, Lits$ where $Lits$ is a set of literals (i.e. negated or non-negated atomic formulae)

▶ axiom

$$\frac{}{P, \Delta, \{\overline{P}\} \cup Lits} \quad (\text{with } P/\overline{P} \text{ unifiable, i.e., } \sigma(P) = \sigma(\overline{P}))$$

▶ α -, β -, and γ -rules

$$\frac{A, \{B\} \cup \Delta, Lits}{A \vee B, \Delta, Lits} \quad \frac{A, \Delta, Lits \quad B, \Delta, Lits}{A \wedge B, \Delta, Lits} \quad \frac{A[x \setminus x^*], \Delta \cup \{\exists x A\}, Lits}{\exists x A, \Delta, Lits}$$

▶ **next-rule** additionally necessary

$$\frac{A, \Delta, \{P\} \cup Lits}{P, \{A\} \cup \Delta, Lits}$$

- ▶ all rules are still **invertible**; P and its complement \overline{P} are literals

Implementing the Block Tableau Calculus

The main predicate is `prove(Fml, UnExp, Lits, FreeV, VarLim)`.

- ▶ **Fml** is the **formula** on the (current) branch that will be considered next
- ▶ **UnExp** is a list of **formulae** on the (current) branch **not expanded** so far
- ▶ **Lits** is a list of **literals** on the (current) branch
- ▶ **FreeV** is a list of **free variables** on the (current) branch
- ▶ **VarLim** specifies the **maximum** number of **free variables** on the branch (used for iterative deepening on the number of free variables on branch)

The **translation** into skolemized negation normal form is done by the predicate `nnf(F, F1)` in the Prolog module `nnf_pure.pl`.

Implementing α -, β - and γ -rule▶ α -, β -, and γ -rules

$$\frac{A, \{B\} \cup \Delta, Lits}{A \vee B, \Delta, Lits} \quad \frac{A, \Delta, Lits \quad B, \Delta, Lits}{A \wedge B, \Delta, Lits} \quad \frac{A[x \setminus x_1], \Delta \cup \{\exists x A\}, Lits}{\exists x A, \Delta, Lits}$$

▶ `leantap_pure.pl`:

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

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

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

Implementing Axiom and Next-rule

▶ axiom

$$\frac{}{P, \Delta, \{\overline{P}\} \cup Lits} \quad (\text{with } P/\overline{P} \text{ unifiable, i.e., } \sigma(P) = \sigma(\overline{P}))$$

▶ `leantap_pure.pl`:

```
prove(Lit, _, Lits, _, _) :-
    (Lit = -Neg; -Lit = Neg) ->
    member(L, Lits), unify1(Neg, L).
```

▶ **next-rule**

$$\frac{A, \Delta, \{P\} \cup Lits}{P, \{A\} \cup \Delta, Lits}$$

▶ `leantap_pure.pl`:

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

leanTAP – A Minimal Tableau Prover

```

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(I,J),A,B,C,D) :- !,
  \+length(C,D),copy_term((I,J,C),(G,F,C)),
  append(A,[all(I,J)],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).

```

- ▶ first popular lean prover [Beckert/Posegga 1995]
- ▶ based on [block tableau calculus](#) for NNF with free variables
- ▶ source code size of minimal version only [360 bytes](#)
- ▶ [performance](#) good on rather easy problems



Term Unification

- ▶ algorithm for term unification according to [Robinson 1965]
 - ▶ $unify(s, t)$ – unification of the terms s and t
 σ – represents [most general unifier](#)
 - ▶ $unify(t, t) \rightarrow \sigma$ remains unchanged
 $unify(x, t) \rightarrow \sigma(x) = t$ if x does not occur in t
 $unify(t, x) \rightarrow \sigma(x) = t$ if x does not occur in t
 $unify(f(s_1, \dots, s_n), f(t_1, \dots, t_n)) \rightarrow unify(s_1, t_1), \dots, unify(s_n, t_n)$
 $unify(f(s_1, \dots, s_n), g(t_1, \dots, t_n)) \rightarrow fail$
- ▶ this algorithm has an [exponential](#) worst-case time complexity
- ▶ there exist algorithms with polynomial time complexity



Implementing Term Unification

- ▶ $unify1(A, B) :- unify([A], [B])$. succeeds iff A and B are unifiable
- ▶ if lists are [empty](#)
 $unify([], [])$.
- ▶ if A and B are [identical terms](#) (e.g. identical variables)
 $unify([A|A1], [B|B1]) :- A==B, !, unify(A1, B1)$.
- ▶ if A/B is a [variable](#) and A/B does not occur in B/A: [assign B/A to A/B](#)
 $unify([A|A1], [B|B1]) :- var(A), !, not_in(A, B), A=B, unify(A1, B1)$.
 $unify([A|A1], [B|B1]) :- var(B), !, not_in(B, A), A=B, unify(A1, B1)$.
 $[not_in(A, B) :- term_variables(B, L), \+ (member(X, L), X==A).]$
- ▶ [otherwise](#), if $A=f(s_1, \dots, s_n)$ and $B=f(t_1, \dots, t_n)$, unify s_i and t_i for $1 \leq i \leq n$
 $unify([A|A1], [B|B1]) :- A=..[F|ArgA], B=..[F|ArgB],$
 $length(ArgA, N), length(ArgB, N), unify(ArgA, ArgB), unify(A1, B1)$.

There is also a Prolog [built-in](#) predicate: $unify_with_occurs_check(A, B)$.



Hands-On: Run the Tableau Prover

- ```

▶ $> swipl % start SWI-Prolog
▶ [leanseq_v5]. % load the sequent prover
▶ [ex_quant]. % load the quant formula
 fof(quant,_,F), prove(F). % and try to prove it
[ex_f12]. % load the f12 formula
 fof(f12,_,F), prove(F). % and try to prove it

▶ [leantap_pure]. % load the tableau prover

▶ [ex_barber]. % load the barber puzzle
 fof(barber,_,F), prove(F). % solve puzzle

▶ [ex_quant]. % load the quant formula
 fof(quant,_,F), prove(F). % and try to prove it
[ex_f12]. % load the f12 formula
 fof(f12,_,F), prove(F). % and try to prove it
[ex_f20]. % load the f20 formula
 fof(f20,_,F), prove(F). % and try to prove it

```



# How to Build an Automated Theorem Prover

## Part 3b: A Connection Prover

Jens Otten

University of Oslo



# Connection Calculus – Motivation

**Example:**  $man(Plato) \wedge \forall x (man(x) \rightarrow mortal(x)) \rightarrow mortal(Plato)$

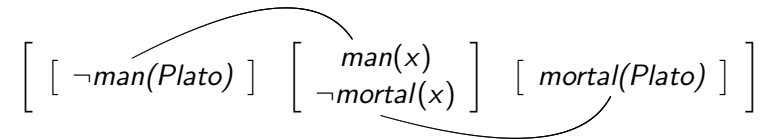
▶ **disjunctive normal form:**

$$\exists x (\neg man(Plato) \vee (man(x) \wedge \neg mortal(x)) \vee mortal(Plato))$$

▶ **matrix** = set of **clauses**:

$$\{\{\neg man(Plato)\}, \{man(x), \neg mortal(x)\}, \{mortal(Plato)\}\}$$

▶ **graphical representation** of matrix:



▶ **connection** is, e.g.,  $\{\neg man(Plato), man(x)\}$  for  $\sigma(x)=Plato$



# Basic Concepts for the Connection Calculus

- ▶ a **matrix** represents a formula in **disjunctive normal form** as set of clauses  $\{\{L_1, L_2, \dots\}, \{L'_1, L'_2, \dots\}, \dots\}$
- ▶ in the **graphical representation** of a matrix, its clauses are arranged horizontally, literals of clauses vertically
- ▶ a **path** through matrix  $M=\{C_1, \dots, C_n\}$  contains one literal from each of its clauses
- ▶ a **connection** is a set of literals of the form  $\{P, \neg P\}$
- ▶ for a substitution  $\sigma$ ,  $\{L_1, L_2\}$  is a  **$\sigma$ -complementary connection** if  $\sigma(L_1)=\sigma(\overline{L_2})$
- ▶ a **multiplicity**  $\mu : M \rightarrow \mathbb{N}$  specifies the number of clauses copies for a matrix  $M$ ;  $M^\mu$  is the matrix that includes these copies



# Matrix Characterization of Logical Validity

## Theorem (Matrix Characterization)

A matrix  $M$  is **valid** (in classical logic) iff there is

- ▶ a **substitution**  $\sigma$ ,
- ▶ a **multiplicity**  $\mu$ ,
- ▶ and a **set of connections**  $S$ ,

such that **every path** through  $M^\mu$  contains a  **$\sigma$ -complementary connection**  $\{L_1, L_2\} \in S$ .

**Proof.**

See [Andrews 1981, Bibel 1981]. □



Tableau Calculus   A Tableau Prover   Term Unification   Connection Calculus   A Connection Prover   Summary

## Matrix Characterization – Examples

$F$  is **valid**  $\Leftrightarrow$  every path through its **matrix** contains a **connection**

- ▶ **matrix**: set of clauses in **disjunctive normal form**
- ▶ **path**: pick one element from each clause
- ▶ **connection**: set of literals of the form  $\{P, \neg P\}$

**Examples:**

- ▶  $p \vee \neg p \rightsquigarrow \left[ \left[ p \right] \left[ \neg p \right] \right] \rightsquigarrow$  **valid**
- ▶  $\neg p \vee (p \wedge \neg q) \vee \neg q \rightsquigarrow \left[ \left[ \neg p \right] \left[ \begin{matrix} p \\ \neg q \end{matrix} \right] \left[ q \right] \right] \rightsquigarrow$  **valid**
- ▶  $(p \wedge q \wedge \neg r) \vee (\neg q \wedge p) \vee \neg p \vee r$   
 $\rightsquigarrow \left[ \left[ \begin{matrix} p \\ q \\ \neg r \end{matrix} \right] \left[ \begin{matrix} \neg q \\ p \end{matrix} \right] \left[ \neg p \right] \left[ r \right] \right] \rightsquigarrow$  **valid**

◀ ▶

Jens Otten (UiO)   How to Build a Theorem Prover — Part 3   TABLEAUX Tutorial 2019 17 / 28

Tableau Calculus   A Tableau Prover   Term Unification   Connection Calculus   A Connection Prover   Summary

## Proof Search in the Connection Calculus

**Proof search** in the connection calculus:

- ▶ check that every path contains a ( $\sigma$ -complementary) **connection**
- ▶ use a **connection-driven** search strategy
- ▶ guided by an **active path** (subset of path) and a **subgoal clause**
- ▶ once a **connection is identified**, all paths containing this connection are excluded from any subsequent investigations
- ▶ calculates the substitution  $\sigma$  using a **term unification** algorithm
- ▶ clauses are **copied** during the proof search (multiplicity  $\mu$ )

◀ ▶

Jens Otten (UiO)   How to Build a Theorem Prover — Part 3   TABLEAUX Tutorial 2019 18 / 28

Tableau Calculus   A Tableau Prover   Term Unification   Connection Calculus   A Connection Prover   Summary

## Solving the Diet Puzzle

**Prove** that the man **always has beer** for dinner.

- ▶ representation as a two-dimensional **matrix** (clausal form):

- ▶ every “**path**” through the matrix contains a **connection**
- ▶ formula/matrix is **valid**, i.e., the man **has always beer** for dinner

◀ ▶

Jens Otten (UiO)   How to Build a Theorem Prover — Part 3   TABLEAUX Tutorial 2019 19 / 28

Tableau Calculus   A Tableau Prover   Term Unification   Connection Calculus   A Connection Prover   Summary

## Connection Calculus – Axiom and Rules

- ▶ **Axiom**  $\frac{}{\{\}, M, Path}$
- ▶ **Start rule**  $\frac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}$   $C_2$  is copy of  $C_1 \in M$
- ▶ **Reduction rule**  $\frac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}}$   $\{\sigma(L_1), \sigma(L_2)\}$  is a **connection**
- ▶ **Extension rule**  $\frac{C_2 \setminus \{L_2\}, M, Path \cup \{L_1\}}{C \cup \{L_1\}, M, Path}$   $C_2$  is copy of  $C_1 \in M$ ,  $L_2 \in C_2$ ,  $\{\sigma(L_1), \sigma(L_2)\}$  is a **connection**
- ▶ a **connection proof** with substitution  $\sigma$  for matrix  $M$  is a proof of  $\varepsilon, M, \varepsilon$  in the connection calculus

◀ ▶

Jens Otten (UiO)   How to Build a Theorem Prover — Part 3   TABLEAUX Tutorial 2019 20 / 28

## Implementing the Start Rule

## ▶ Start rule

$$\frac{C_2, M, \{\}}{\varepsilon, M, \varepsilon} \quad C_2 \text{ is copy of } C_1 \in M$$

- ▶ an additional argument is added that contains the (path size) limit for **iterative deepening**
- ▶ the module `nnf_mm.pl` contains the predicate `make_matrix(F,M)` for **translating** a first-order formula F into a **matrix** M
- ▶ `leancop_pure.pl`:

```
prove(M) :- M=[_|_] -> prove(M,1) ;
 make_matrix(M,M1), prove(M1,1).

prove(M,I) :- print(iteration:I), nl,
 member(C1,M), copy_term(C1,C2),
 prove(C2,M, [], I).
prove(M,I) :- I1 is I+1, prove(M,I1).
```



## Implementing Axiom and Reduction Rule

## ▶ Axiom

$$\frac{}{\{\}, M, Path}$$

▶ `leancop_pure.pl`:

```
% axiom
prove([],_,_,_).
```

## ▶ Reduction rule

$$\frac{C, M, Path \cup \{L_2\}}{CU\{L_1\}, M, Path \cup \{L_2\}} \quad \{\sigma(L_1), \sigma(L_2)\} \text{ is a connection}$$

▶ `leancop_pure.pl`:

```
% reduction
prove([L1|C],M,Path,I) :- (L1= -N1; -L1=N1) ->
 member(L2,Path), unify1(N1,L2),
 prove(C,M,Path,I).
```



## Implementing the Extension Rule

## ▶ Extension rule

$$\frac{C_2 \setminus \{L_2\}, M, Path \cup \{L_1\}}{CU\{L_1\}, M, Path} \quad C_2 \text{ is copy of } C_1 \in M, L_2 \in C_2, \{\sigma(L_1), \sigma(L_2)\} \text{ is a connection}$$

▶ `leancop_pure.pl`:

```
% extension
prove([L1|C],M,Path,I) :- \+ length(Path,I),
 (L1= -N1; -L1=N1) ->
 member(C1,M), copy_term(C1,C2),
 select1(L2,C2,C3), unify1(N1,L2),
 prove(C3,M, [L1|Path], I),
 prove(C,M,Path,I).
```



## leanCoP 1.0 – A Minimal Connection Prover

```
prove(M,I) :- append(Q, [C|R], M), \+member(-_,C),
 append(Q,R,S), prove([!], [[-!|C]|S], [], I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(N,P);
 append(Q, [D|R], M), copy_term(D,E), append(A, [N|B], E),
 append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
 append(R, [D|Q], S)), prove(F,S, [L|P], I)), prove(C,M,P,I).
```

- ▶ based on (clausal) **connection calculus**; **CoP=Connection Prover**
- ▶ size of minimal (Prolog) source code is **333 bytes**
- ▶ start rule restricted to **“positive” clauses**
- ▶ clause copies and path limit check only for **non-ground clauses**
- ▶ **sound & complete**; decision procedure for propositional logic
- ▶ surprisingly **good performance**



## leanCoP 2.0 – An Improved Connection Prover

- ▶ leanCoP v1.0 implements basic calculus
- ▶ leanCoP v2.0 integrates optimizations (minimal code: 555 bytes):

```

prove(I,S) :- \+member(scut,S) -> prove([-(#)], [], I, [], S) ;
 lit(#,C,_) -> prove(C,[-(#)], I, [], S).
prove(I,S) :- member(comp(L),S), I=L -> prove(1, []) ;
 (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,_).
prove([L|C],P,I,Q,S) :- \+ (member(A,[L|C]), member(B,P),
 A==B), (-N=L;-L=N) -> (member(D,Q), L=D ;
 member(E,P), unify_with_occurs_check(E,N) ; lit(N,F,H),
 (H=g -> true ; length(P,K), K<I -> true ;
 \+p -> assert(p), fail), prove(F,[L|P],I,Q,S)),
 (member(cut,S) -> ! ; true), prove(C,P,I,[L|Q],S).

```

## Hands-On: Run the Connection Prover

- ▶ `$> swipl` % start SWI-Prolog
- ▶ `[leancop_pure].` % load the connection prover
- ▶ `[ex_barber].` % load the barber puzzle  
`fof(barber,_,F), prove(F).` % solve puzzle
- ▶ `prove( ![X]:p(X) => p(a) & p(b) ).`  
`prove( ![X]: p(X) => ?[Y]: p(Y) ).`  
`prove( ?[X]: p(X) => ![Y]: p(Y) ).`
- ▶ `[ex_quant].` % load the quant formula  
`fof(quant,_,F), prove(F).` % and try to prove it
- ▶ `[ex_f12].` % load the f12 formula  
`fof(f12,_,F), prove(F).` % and try to prove it
- ▶ `[ex_f20].` % load the f20 formula  
`fof(f20,_,F), prove(F).` % and try to prove it
- ▶ `halt.` % exit SWI-Prolog

## Summary and Outlook

- ▶ **general approach** to translate a calculus into a Prolog program
- ▶ **Prolog program** implements the (proof) search in an elegant way
- ▶ key techniques for (classical) **propositional** logic:
  - ▶ **bottom-up** proof search
  - ▶ identify **invertible rules** and cut the search space
- ▶ key techniques for (classical) **first-order** logic:
  - ▶ **free** variables
  - ▶ **iterative** deepening
  - ▶ (dynamic) **skolemization**
- ▶ techniques are extended to **tableau** and **connection calculi**
  - ▶ **skolemized NNF**
  - ▶ **connection-driven** proof search

## Summary and Outlook

- ▶ the presented techniques can also be used to implement **other** proof calculi, e.g., for **non-classical** logics
- ▶ **for example**, the rules for negation of the **intuitionistic** (propositional multi-succedent) sequent calculus

$$\frac{\Gamma_1, \neg A \Rightarrow A, \Delta}{\Gamma_1, \neg A \Rightarrow \Delta} \neg\text{-left} \qquad \frac{\Gamma, A \Rightarrow}{\Gamma \Rightarrow \neg A, \Delta_1} \neg\text{-right}$$

can easily be implemented in **Prolog**

```

prove(G > D) :- member(~A,G), prove(G > [A|D]).
prove(G > D) :- select1(~A,D,_), prove([A|G] > []).

```

- ▶ **play** around, ... extend, improve, adapt, ... **have fun**