Implementing ATP Systems

Unit 9: Basic Calculi

Jens Otten

University of Potsdam

Outline

Negation Normal Form

- Is the following formula valid in classical logic?
  $(\,((\exists x Q(x) \vee \neg Q(c)) \Rightarrow P) \wedge (P \Rightarrow (\exists y Q(y) \wedge R))\,) \Rightarrow (P \wedge R)$
- Removing equivalences/implications; moving negation inside:
  $((\exists x Q(x) \vee \neg Q(c)) \wedge \neg P) \vee (P \wedge (\forall y \neg Q(y) \vee \neg R)) \vee (P \wedge R)$
- Removing universal quantifiers (by Skolemization):
  $((\exists x Q(x) \vee \neg Q(c)) \wedge \neg P) \vee (P \wedge (\neg Q(b) \vee \neg R)) \vee (P \wedge R)$
- Negating formula (unsatisfiable iff original formula is valid):
  $((\forall x \neg Q(x) \wedge Q(c)) \vee P) \wedge (\neg P \vee (Q(b) \wedge R)) \wedge (\neg P \vee \neg R)$
- Representation in Prolog:
  `((all(X,-q(X)),q(c));p) , (-p;(q(b),r)) , (-p;-r)`

leanTAP: A Lean 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).
```

- Based on analytic tableaux with free variables.
- `prove(Fml,[],[],[],VarLim)` succeeds iff there is a tableau for `Fml` with at most `VarLim` free variables on each branch.
- Source code size of minimal version only 360 bytes.
- Requires (only) negation normal form.

Lean Theorem Proving

What is "lean theorem proving"?

- Compact source code.
- Elegant implementation techniques.
- Basic calculus + some essential search heuristics.
- Considerable performance by using extremely compact code.
- In general implemented in Prolog.
- Important: "lean" $\neq$ "simple".

First popular lean prover: leanTAP (Beckert/Posegga '95).

- Based on analytic tableaux with free variables.
- But performance on more difficult problems rather poor.

Disjunctive Normal Form and Clausal Form

- Is the following formula valid in classical logic?
  $(((\exists x Q(x) \vee \neg Q(c)) \Rightarrow P) \wedge (P \Rightarrow (\exists y Q(y) \wedge R))) \Rightarrow (P \wedge R)$
- Translation into disjunctive normal form (b is Skolem term):
  $(P \wedge R) \vee (\neg P \wedge Qx) \vee (\neg Qb \wedge P) \vee (\neg Qc \wedge \neg P) \vee (P \wedge \neg R)$
- Representation as set of clauses (= matrix):
  $\{ \{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \{\neg Qc, \neg P\}, \{P, \neg R\} \}$
- Representation as graphical matrix:

$$\left[\begin{bmatrix} P \\ R \end{bmatrix} \begin{bmatrix} \neg P \\ Qx \end{bmatrix} \begin{bmatrix} \neg Qb \\ P \end{bmatrix} \begin{bmatrix} \neg Qc \\ \neg P \end{bmatrix} \begin{bmatrix} P \\ \neg R \end{bmatrix} \right]$$

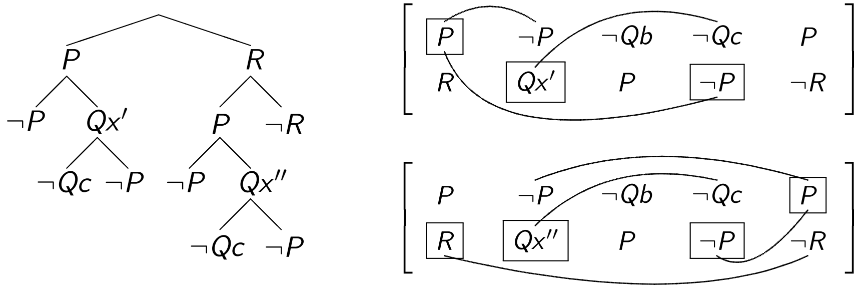
- Representation of matrix in Prolog:

```
[[p,r], [-p,q(X)], [-q(b),p], [-q(c),-p], [p,-r]]
```

Example: Connection Calculus

- Is the following matrix valid in classical logic?

  $\{\,\{P,R\},\{\neg P,Qx\},\{\neg Qb,P\},\{\neg Qc,\neg P\},\{P,\neg R\}\,\}$



- Answer: Matrix is valid in classical logic (with $\sigma(x')=\sigma(x'')=c$).

Connection Calculus: Formal Representation

- Axiom $\dfrac{}{\{\}, M, Path}$

- Start Rule $\dfrac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}$ 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

$$\frac{C_2 \setminus \{L_2\}, M, Path \cup \{L_1\} \qquad C, M, Path}{C \cup \{L_1\}, M, Path}$$

C_2 is copy of $C_1 \in M$, $L_2 \in C_2$,
$\{\sigma(L_1), \sigma(L_2)\}$ is a connection

- Connection proof
  $\Leftrightarrow \exists$ derivation for $\varepsilon, M, \varepsilon$ in which all leaves are axioms.

Example: Formal Connection Calculus

- Is the following matrix valid in classical logic?

  $M = \{\, \{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \{\neg Qc, \neg P\}, \{P, \neg R\} \,\}$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\}, M, \{P, \neg Qx'\}}\,a}{\{\neg P\}, M, \{P, \neg Qx'\}}\,r \quad \dfrac{}{\{\}, M, \{P\}}\,a}{\{Qx'\}, M, \{P\}}\,e \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\}, M, \{R, P, Qx''\}}\,a}{\{\neg P\}, M, \{R, P, Qx''\}}\,r \quad \dfrac{}{\{\}, M, \{R, P\}}\,a}{\{Qx''\}, M, \{R, P\}}\,e \quad \dfrac{}{\{\}, M, \{R\}}\,a}{\{P\}, M, \{R\}}\,e \quad \dfrac{}{\{\}, M, \{\}}\,a}{\{R\}, M, \{\}}\,e}{\{P, R\}, M, \{\}}\,e}{\varepsilon, \{\{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \{\neg Qc, \neg P\}, \{P, \neg R\}\}, \varepsilon}\; \textit{start rule}
$$

(e: extension rule; r: reduction rule; a: axiom)

- Answer: Matrix is valid in classical logic (with $\sigma(x') = \sigma(x'') = c$).

Implementations of Connection Calculi

- Connection calculi (connection-driven proof search), e.g.
  - model elimination (Loveland '68),
  - connection method (Bibel '83),
  - connection tableau calculus (Letz '94).

- PTTP (Stickel '88): Prolog Technology Theorem Prover.
- METEORs (Astrachan/Loveland '91).
- SETHEO (Letz et al. '92): SEquential THEOrem prover.
- KoMeT (Bibel et al. '94).

- leanCoP v1.0 (Otten/Bibel '03); minimal code: 333 bytes.
- leancCoP v2.0 (Otten '08); minimal code: 555 bytes.

leanCoP: Code and Features

```
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).
```

- leanCoP 1.0 (Otten/Bibel '03); source code just 333 bytes.
- Based on (clausal) connection calculus.
- Motivation: provide students with compact implementation.
- Sound & complete, decision procedure for propositional logic.
- Comparatively strong performance.

Implementation: Start Rule

```
prove(Mat,PathLim) :-
     append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
     append(MatA,MatB,Mat1),
     prove([!],[[-!|Cla]|Mat1],[],PathLim).
```

- `prove(Mat,PathLim)` succeeds iff there is a proof for the matrix `Mat` whose active path length is limited by `PathLim`.
- Select positive start clause `Cla` using "append technique" (every valid matrix contains at least one positive clause!).

- `prove(Subgoals,Mat,Path,PathLim)` succeeds iff there is a proof for `Subgoal` using matrix `Mat` and active path `Path` whose active path length is limited by `PathLim`.
- Start with subgoal "`!`" and add "`-!`" to original start clause (necessary as only clauses in `Mat` are copied).

Implementation: "Append Technique"

- Prolog predicate "append" usually used to append two lists.
  Example: `append([a],[b,c],L)` ⇝ `L=[a,b,c]`

- If first two arguments are (uninstantiated) variables and last argument is a list, all possible solutions to append a list by using the first two arguments are given on backtracking.
  Example:
  ```
  ?- append(A,[X|B],[a,b,c]), append(A,B,C).
  ```
  ⇝
  ```
  A=[]   , X=a, B=[b,c], C=[b,c] ;
  A=[a]  , X=b, B=[c]  , C=[a,c] ;
  A=[a,b], X=c, B=[]   , C=[a,b]
  ```

- "`append(A,[X|B],L), append(A,B,L1)`" elegant way to select literal `X` from list `L` and return list `L1` without `X`.

Implementation: Axiom and Extension/Reduction Rule

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

- prove([],_,_,_) succeeds iff subgoal list is empty.

```
prove([Lit|Cla],Mat,Path,PathLim) :-
     (-NegLit=Lit;-Lit=NegLit) ->
        % extension and/or reduction step
        prove(Cla,Mat,Path,PathLim).
```

- prove(...) succeeds iff there is a proof for [Lit|Cla] (using clauses in Mat, active Path and path limit PathLim).
- NegLit is bound to negated literal of Lit.
- Perform extension and/or reduction step using literal NegLit and continue to prove remaining subgoal list Cla.

Implementation: Extension Rule I (Propositional)

```
prove([Lit|Cla],Mat,Path,PathLim) :-
      (-NegLit=Lit;-Lit=NegLit) ->
         ( append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
           append(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
           append(MatB,MatA,Mat1),
           prove(Cla3,Mat1,[Lit|Path],PathLim)
         ),
         prove(Cla,Mat,Path,PathLim).
```

- Select `Cla1` from clause set `Mat` and copy `Cla1` (= `Cla2`).
- Search for `NegLit` in clause copy `Cla2`.
- `Cla3` is new subgoal list, `Mat1` is new clause set.
- Prove new subgoal list `Cla3` with clauses in `Mat1` and add the literal `Lit` to active path `Path`.

Implementation: Extension Rule II (First-Order)

```
prove([Lit|Cla],Mat,Path,PathLim) :-
     (-NegLit=Lit;-Lit=NegLit) ->
        ( append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
          append(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
          ( Cla1==Cla2 ->
               append(MatB,MatA,Mat1)
               ;
               length(Path,K), K<PathLim,
               append(MatB,[Cla1|MatA],Mat1)
          ),
          prove(Cla3,Mat1,[Lit|Path],PathLim)
        ),
        prove(Cla,Mat,Path,PathLim).
```

- If Cla1 contains no (first-order) variable, Cla1==Cla2 holds.
- Otherwise check if length K of current Path exceeds limit PathLim and include copied clause Cla1 in clause set Mat1.

Implementation: Reduction Rule

```
prove([Lit|Cla],Mat,Path,PathLim) :-
     (-NegLit=Lit;-Lit=NegLit) ->
        ( member(NegLit,Path)
          ;
          append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
          append(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
          ( Cla1==Cla2 ->
               append(MatB,MatA,Mat1)
               ;
               length(Path,K), K<PathLim,
               append(MatB,[Cla1|MatA],Mat1)
          ),
          prove(Cla3,Mat1,[Lit|Path],PathLim)
        ),
        prove(Cla,Mat,Path,PathLim).
```

- Try to apply reduction rule first, i.e. check whether literal `NegLit` is an element of the active path `Path`.

Implementation: Iterative Deepening

```
prove(Mat,PathLim) :-
     append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
     append(MatA,MatB,Mat1),
     prove([!],[[-!|Cla]|Mat1],[],PathLim).

prove(Mat,PathLim) :-
     nonground(Mat), PathLim1 is PathLim+1, prove(Mat,PathLim1).
```

- If set of clauses Mat is not ground (i.e. it contains variables), the path limit PathLim is increased and proof search restarted (necessary to achieve completeness for first-order logic).
- If Mat contains no variables, proof search ends; this yields a decision procedure for propositional logic (remember that PathLim is only checked for clauses containing variables).

Motivation: ATP in Non-Classical Logics

- Constructive/Intuitionistic logic used when formalising computation (proof-as-programs, e.g. in NuPRL, Coq).
- Modal logic used within formal verification (of, e.g., circuits, protocols, programs).
- Linear logic used when reasoning about action and change (e.g. modeling concurrent computation).

- Many calculi and high-performance ATP systems available for classical (clausal!) logic (e.g. Otter, Setheo, E, Vampire).
- But only few high-performance ATP systems available for first-order intuitionistic/modal/linear logic.

Using Prefixes for Intuitionistic Logic

- Prefix is a string over a set of variables $\mathcal{V}$ (capital letters) and constants $\mathcal{C}$ (small letters) assigned to each atomic formula; it specifies the position of atomic formulae within the formula.
- Semantic view: Encode (Kripke-)semantics by embedding into the modal logic S4.
- Proof theoretical view: Encode non-permutabilities of critical rules in the intuitionistic (Fitting-style) sequent calculus.

  Application of critical rules $\Rightarrow$-right, $\neg$-right, $\forall$-right, i.e.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B, \Delta} \qquad \frac{\Gamma, A \vdash}{\Gamma \vdash \neg A, \Delta} \qquad \frac{\Gamma \vdash A[x \backslash a]}{\Gamma \vdash \forall x A, \Delta}$$

  is represented by a constant in the prefix.

Example: Using Prefixes for Intuitionistic Logic

- Application of critical rules $\Rightarrow$-right, $\neg$-right, $\forall$-right.

1. $P \Rightarrow P$ $\qquad \dfrac{P \vdash P}{\vdash P \Rightarrow P}$

2. $P \vee \neg P$ $\qquad \dfrac{\dfrac{P \vdash}{\vdash P, \neg P}}{\vdash P \vee \neg P}$

- Prefix of atomic formula P: sequence of variable/constant for every $\Rightarrow$,$\neg$,$\forall$ with polarity 1/0 preceding P in the formula.

- Intuitionistic substitution σ_J: $\mathcal{V} \mapsto (\mathcal{C} \cup \mathcal{V})^*$ has to unify the prefixes of atomic formulae in every connection.

1. $P \Rightarrow P$ $\quad [\ P^1 : c_0 C_1 \quad P^0 : c_0 c_2\ ]$, $c_0 C_1 = c_0 c_2 \rightarrow \sigma_J(C_1) = c_2 \rightsquigarrow$ valid.
2. $P \vee \neg P$ $\quad [\ P^0 : c_0 \quad P^1 : c_1 C_2\ ]$, $c_0 \neq c_1 C_2 \rightsquigarrow$ not valid.

Matrix Characterisation of Intuitionistic Validity

- Matrix characterisation: F intuitionistically valid $\Leftrightarrow$
  1. $\exists$ set of connections $\{\{P^0 x_1 : p_1, P^1 y_1 : q_1\}, ...\}$
  2. $\exists$ first-order substitution $\sigma_Q = \{x_1 \backslash ..., y_1 \backslash ..., ...\}$
  3. $\exists$ intuitionistic substitution $\sigma_J = \{p_1 \backslash ..., q_1 \backslash ..., ...\}$

  so that every path (branch) through the matrix of F^μ contains a connection (axiom) with $\sigma_Q(x_i){=}\sigma_Q(y_i)$ and $\sigma_J(p_i){=}\sigma_J(q_i)$ for some multiplicity μ and admissible σ_Q and σ_J.

- Calculus/Proof search:
  1. Path checking, e.g. sequent calculus (Gentzen), tableau calculus (Fitting), connection calculus (Bibel).
  2. Term unification, e.g. Robinson, Martelli/Montanari.
  3. Prefix unification, e.g. Ohlbach, Otten/Kreitz.

Connection Calculus for Intuitionistic Logic

- Constructive/intuitionistic logic used, e.g., when formalizing computation (proof-as-programs, e.g. in NuPRL, Coq).
- Idea: Extend the classical prover leanCoP based on a clausal connection calculus for intuitionistic logic (Otten '05) using prefixes (Wallen '90).
- Connection-based proof search:
  - During clausal form translation, a prefix(-string) is added to each atomic formula.
  - Skolemization extended to prefix variables and constants.
  - Afterwards, a classical proof search is performed.
  - During the proof search all prefixes of connections are collected.
  - After classical proof is found the collected prefixes are unified by a prefix unification algorithm (e.g. Otten/Kreitz).

ileanCoP: Extending leanCoP

```
prove(M,I) :- append(Q,[G:C|R],M), \+member(-(_):_,C),
 append(Q,R,S), prove([!:[]],[G:[-(!):(-[])|C]|S],[],I,[Y,T]),
 addco(T,!), pruni(Y).
prove([],_,_,_,[[],[]]).
prove([L:U|C],M,P,I,[Y,T]) :- (-N=L; -L=N) -> (member(N:V,P),
 \+ \+ pruni([U=V]), W=[], O=[]; append(Q,[D|R],M),
 copy_term(D,G:E), append(A,[N:V|B],E), \+ \+ pruni([U=V]),
 append(A,B,F), (D==G:E -> append(R,Q,S); length(P,K), K<I,
 append(R,[D|Q],S)), prove(F,S,[L:U|P],I,[W,H]), append(H,G,O)),
 prove(C,M,P,I,[X,J]), append([U=V|W],X,Y), append(J,O,T).
```

- Add prefix P to each literal L, i.e. L:P.
- Collect prefix equations and do prefix unification at the end.
- Add set of variables V to each clause C, i.e. V:C.
- Collect variables and check additional condition at the end.
- Source code size is 524 bytes (without prefix unification).

ileanCoP: Prefix Unification Code

```
pruni([]). pruni([S=T|G]) :- (-X=S -> Y=T ; -X=T, Y=S),
           flatten([X,_],U), flatten(Y,V), tuni(U,[],V), pruni(G).
```

```
tuni([],[],[]).        tuni([],[],[X|T]) :- tuni([X|T],[],[]).
tuni([X|S],[],[Y|T]) :- (var(X) -> (var(Y), X==Y);
                          (\+var(Y), X=Y)), !, tuni(S,[],T).
tuni([C|S],[],[V|T]) :- \+var(C), !, var(V), tuni([V|T],[],[C|S]).
tuni([V|S],Z,[])       :- V=Z, tuni(S,[],[]).
tuni([V|S],[],[C|T]) :- \+var(C), V=[], tuni(S,[],[C|T]).
tuni([V|S],Z,[C,D|T]):- \+var(C), \+var(D), append(Z,[C],V),
                          tuni(S,[],[D|T]).
tuni([V,X|S],[],[W|T])     :- var(W), tuni([W|T],[V],[X|S]).
tuni([V,X|S],[Y|Z],[W|T]) :- var(W), append([Y|Z],[N],V),
 tuni([W|T],[N],[X|S]).    tuni([V|S],Z,[X|T]) :-
      (S=[]; T\=[]; \+var(X)) -> append(Z,[X],Y), tuni([V|S],Y,T).
```

```
addco(X,_)           :- (atomic(X); var(X); X==[[]]), !.
addco([[X,V]|L],!) :- !, addco(X,V), addco(L,!).
addco(_^_^U,V)       :- !, pruni([-U=V]).
addco(T,V)           :- T=..[_,S|R], !, addco(S,V), addco(R,V).
```

Other Lean Theorem Provers

- leanTAP: tableau, classical first-order logic.
- ModLeanTAP: tableau, propositional modal logic.
- λleanTAP: tableau, higher-order logic.
- ileanTAP: tableau + prefixes, intuitionistic first-order logic.
- linTAP: tableau + prefixes, linear logic M?LL.
- leanCoP: connection, classical first-order logic.
- ileanCoP: connection + prefixes, intuitionistic first-order logic.
- lolliCoP: reimplementation of leanCoP in the linear logic programming language Lolli (Hodas/Tamura 2001).
- ncDP: non-clausal DPLL, classical propositional logic.

An Implementation of the DPLL Calculus

```
dpll([])  :- !, fail.
dpll(Mat) :- member([],Mat), !.
dpll(Mat) :-
    Mat=[[Lit|_]|_], (-NegLit=Lit;-Lit=NegLit) ->
    reduce(Mat,Lit,NegLit,Mat1), dpll(Mat1),
    reduce(Mat,NegLit,Lit,Mat2), dpll(Mat2).

reduce([],_,_,[]).
reduce([Cla|Mat],Lit,NegLit,Mat1) :-
    ( member(Lit,Cla) ->
          Mat1=Mat2 ;
          delete(Cla,NegLit,Cla2), Mat1=[Cla2|Mat2]
    ), reduce(Mat,Lit,NegLit,Mat2).
```

- Based on the Davis-Putnam-Logemann-Loveland (DPLL) decision procedure for propositional logic.
- `dpll(Mat)` succeeds iff set of clauses `Mat` is valid.
- `reduce(Mat,Lit,NegLit,Mat1)` returns `Mat1`, in which all clauses that contain `Lit` and all literals `NegLit` are deleted.
- Source code size: 399 bytes (see also `www.leancop.de/ncdp`).

High-Performance Theorem Provers

Based on resolution/paramodulation:

- E: best all-purpose theorem prover (Schulz '02).
- Vampire: most powerful prover (Riazanov/Voronkov '02).
- Otter: the classical resolution prover (McCune '90).
- Prover9: successor of Otter (McCune '08).
- SPASS: using superposition and sorts (Weidenbach et al '07).
- SNARK: SRI's standard theorem prover (Stickel '08).

Instance-based methods (DPLL for first-order logic):

- iProver: fastest instance-based prover (Korovin '08).
- Equinox: successful prover (Claessen '05).
- Darwin: based on model evolution (Baumgartner et al '06).