Build Your Own First-Order Prover

Part 2a: Implementing a Propositional Prover

Jens Otten

University of Oslo

Hands-On: "Diet Puzzle"

"What is the secret of your long life?" a man was asked.

He replied: "I strictly follow my diet: If I don't drink beer for dinner, then I always have fish. Any time I have both beer and fish for dinner, then I do without ice cream. If I have ice cream or don't have beer, then I never eat fish."

Question 1: "Does the man have beer for dinner?"
Question 2: "Does he have both ice cream and fish for dinner?"

- formalize this puzzle in the language of (propositional) logic!
- $($ $(\neg \text{beer} \rightarrow \text{fish})$
  $\wedge \ (\text{beer} \wedge \text{fish} \rightarrow \neg \text{ice\_cream})$
  $\wedge \ (\text{ice\_cream} \vee \neg \text{beer} \rightarrow \neg \text{fish}) \ )$
  $\rightarrow \ \text{beer}$ (for Question 1)
  $\rightarrow \ (\text{ice\_cream} \wedge \text{fish})$ (for Question 2)

Representing First-Order Logic in Prolog

- terms $(f(a,x))$ are represented by Prolog terms (`f(a,X)`) , i.e.,

  variables (x, y, z) are represented by Prolog variables (`X`, `Y`, `Z`) starting with a capital letter;

  constants (a, b, c) are represented by Prolog constants (`a`, `b`, `c`) starting with a small letter
- atomic formulae $(p,\ q(a,x))$ are represented by Prolog terms, i.e., propositional variables and predicate symbols are represented by Prolog constants (`p`, `q(a,X)`) starting with a small letter
- logical operators $\neg, \wedge, \vee, \rightarrow, \forall x, \exists x$ are represented by $\sim$, `&` , `|` , `=>` , `![X]:` , `?[X]:`
- in a Prolog file, a formula F is represented (in TPTP syntax) by `fof(`*formula_name*`,conjecture,`F`).`

Hands-On: Representing Logic in Prolog

- download SWI-Prolog at `https://www.swi-prolog.org`
- open your favourite editor and express the "diet puzzle" in Prolog syntax and save it in files named `ex_diet1.pl` and `ex_diet2.pl`
- `ex_diet1.pl`:

```
fof(diet1,conjecture,
(~b => f) & ((b & f) => ~i) & ((i | ~b) => ~f) => b ).
```

- `ex_diet2.pl`:

```
fof(diet2,conjecture,
(~b => f) & ((b & f) => ~i) & ((i | ~b) => ~f) => (i&f) ).
```

- you can find the code for this example and all of the following Prolog source code on the tutorial's website at

  `http://jens-otten.de/tutorial_cade19`

How to Implement a Calculus in Prolog?

Remember: A (proof) calculus consists of

- axioms of the form $\frac{}{w}$
- rules of the form $\frac{w_1 \quad w_2 \quad \cdots \quad w_n}{w}$

($w_1, \ldots, w_n$ are the premises, w is the conclusion)

It can be implemented in Prolog in the following way:

- axioms: `prove(w).`
- rules: `prove(w) :- prove(w1), prove(w2), ..., prove(wn).`
- query "is there a proof for w'?": `?- prove(w').`

Proof search (with backtracking) is done (implicitly) by Prolog.

- for a complete proof search, iterative deepening has to be added
- first optimization: bottom-up proof search

Representing Sequents in Prolog

Remember: A sequent has the form

$$\Gamma \implies \Delta \qquad \text{with} \quad \Gamma = \{A_1, \ldots, A_n\}, \ \Delta = \{B_1, \ldots, B_m\}$$

and is represented in Prolog by two lists:

$$\Gamma \texttt{ > } \Delta \qquad \text{with} \quad \Gamma = \texttt{[A1,...,An]}, \ \Delta = \texttt{[B1,...,Bm]}$$

- the predicate `select1(A,G,G')` succeeds if `A` is element of list `G` and returns `G'` as `G` without `A`, e.g. `select1(b,[a,b,c],L)` $\rightsquigarrow$ `L=[a,c]`
- a rule of the form $\frac{A', \Gamma' \implies \Delta}{A, \Gamma' \implies \Delta}$ with $\Gamma = \{A\} \cup \Gamma'$ is implemented by

```
prove(G > D) :- select1(A,G,G'), prove([A'|G'] > D).
```

- the proof search of formula A is invoked by `?- prove([] > [A]).`
- implementation of the `select1` predicate:
  `select1(X,L,L1) :- append(L2,[X|L3],L), append(L2,L3,L1).`

Hands-On: Getting Started

- definition of the logical operators and the select1 predicate
- new operators are defined with: `op(Precedence,Type,Name)`
- `leanseq_v0.pl`:

```
:- op( 500, fy, ~).     % negation
:- op(1000, xfy, &).    % conjunction
:- op(1100, xfy, '|').  % disjunction
:- op(1110, xfy, =>).   % implication
:- op( 500, fy, !).     % universal quantifier:  ![X]:
:- op( 500, fy, ?).     % existential quantifier:  ?[X]:
:- op( 500,xfy, :).
% ----------------------------------------------------------
prove0(F) :- prove([] > [F]).
% ----------------------------------------------------------
% some space for the actual prover
% ----------------------------------------------------------
select1(X,L,L1) :- append(L2,[X|L3],L), append(L2,L3,L1).
% ----------------------------------------------------------
```


Sequent Calculus – Axiom

- the only axiom

$$\frac{}{\Gamma_1, A \implies A, \Delta_1} \text{ axiom}$$

- `leanseq_v1.pl`:

```
% axiom
prove(G > D) :- member(A,G), member(A,D).
```

Sequent Calculus – Rules for $\wedge$

- rules for $\wedge$ (conjunction)

$$\frac{\Gamma_1, A, B \implies \Delta}{\Gamma_1, A \wedge B \implies \Delta}\ \wedge\text{-left} \qquad \frac{\Gamma \implies A, \Delta_1 \quad \Gamma \implies B, \Delta_1}{\Gamma \implies A \wedge B, \Delta_1}\ \wedge\text{-right}$$

- leanseq_v1.pl:

```
% conjunction
prove(G > D) :- select1(A&B,G,G1),
                prove([A,B|G1] > D).

prove(G > D) :- select1(A&B,D,D1),
                prove(G > [A|D1]), prove(G > [B|D1]).
```


Sequent Calculus – Rules for $\vee$

- rules for $\vee$ (disjunction)

$$\frac{\Gamma_1, A \implies \Delta \quad \Gamma_1, B \implies \Delta}{\Gamma_1, A \vee B \implies \Delta}\ \vee\text{-left} \qquad \frac{\Gamma \implies A, B, \Delta_1}{\Gamma \implies A \vee B, \Delta_1}\ \vee\text{-right}$$

- leanseq_v1.pl:

```
% disjunction
prove(G > D) :- select1(A|B,G,G1),
                prove([A|G1] > D), prove([B|G1] > D).

prove(G > D) :- select1(A|B,D,D1),
                prove(G > [A,B|D1]).
```


Sequent Calculus – Rules for $\rightarrow$

- rules for $\rightarrow$ (implication)

$$\frac{\Gamma_1 \implies A, \Delta \quad \Gamma_1, B \implies \Delta}{\Gamma_1, A \rightarrow B \implies \Delta}\ \rightarrow\text{-left} \qquad \frac{\Gamma, A \implies B, \Delta_1}{\Gamma \implies A \rightarrow B, \Delta_1}\ \rightarrow\text{-right}$$

- leanseq_v1.pl:

```
% implication
prove(G > D) :- select1(A=>B,G,G1),
                prove(G1 > [A|D]), prove([B|G1] > D).

prove(G > D) :- select1(A=>B,D,D1),
                prove([A|G] > [B|D1]).
```


Sequent Calculus – Rules for $\neg$

- rules for $\neg$ (negation)

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

- leanseq_v1.pl:

```
% negation
prove(G > D) :- select1(~A,G,G1),
                prove(G1 > [A|D]).

prove(G > D) :- select1(~A,D,D1),
                prove([A|G] > D1).
```

The Complete Prover for Propositional Logic

```
prove0(F) :- prove([] > [F]).
prove(G > D) :- member(A,G), member(A,D).                % axiom
prove(G > D) :- select1(A&B,G,G1),                       % conjunction
                prove([A,B|G1] > D).
prove(G > D) :- select1(A&B,D,D1),
                prove(G > [A|D1]), prove(G > [B|D1]).
prove(G > D) :- select1(A|B,G,G1),                       % disjunction
                prove([A|G1] > D), prove([B|G1] > D).
prove(G > D) :- select1(A|B,D,D1),
                prove(G > [A,B|D1]).
prove(G > D) :- select1(A=>B,G,G1),                      % implication
                prove(G1 > [A|D]), prove([B|G1] > D).
prove(G > D) :- select1(A=>B,D,D1),
                prove([A|G] > [B|D1]).
prove(G > D) :- select1(~A,G,G1),                        % negation
                prove(G1 > [A|D]).
prove(G > D) :- select1(~A,D,D1),
                prove([A|G] > D1).
select1(X,L,L1) :- append(L2,[X|L3],L), append(L2,L3,L1).
```


Hands-On: Run Your Prover

- `$> swipl` % start SWI-Prolog
- `[leanseq_v1].` % load your prover
  `make.` % reload prover after changing the source code
- `prove0(a=>a).`
  `prove0(hello).`
  `prove0(to_be | ~to_be).`
  `prove0(to_be & ~to_be).`
  `prove0(p & (p => q) => q).`
- `[ex_diet1].` % load the diet puzzle (question 1)
  `fof(diet1,_,F).` % check the loaded formula
  `fof(diet1,_,F), prove0(F).` % solve the 1st question
- `[ex_diet2].` % load the diet puzzle (question 2)
  `fof(diet2,_,F).` % check the loaded formula
  `fof(diet2,_,F), prove0(F).` % solve the 2nd question
- `halt.` % exit SWI-Prolog

Invertible Rules

- in general all applicable (alternative) rules of a calculus have to be applied during the proof search (on backtracking)
- e.g., for "$(p \wedge q), (q \rightarrow r) \implies r$", $\wedge$-left and $\rightarrow$-left can be applied
- order of rule applications impacts search space and provability
- a rule $\frac{w_1 \quad w_2 \quad \cdots \quad w_n}{w}$ is invertible, if whenever w is provable (valid) so are $w_1, w_2, ..., w_n$ (opposite direction obviously holds)
- in case of invertible rules, no alternative rules need to be applied, i.e. proof search can be cut off, resulting in smaller search space
- all rules of the presented propositional calculus are invertible
- e.g., the rule $\frac{\Gamma \implies A, B, \Delta}{\Gamma \implies A \vee B, \Delta}$ $\vee$-right is invertible
- example for non-invertible rule is $\frac{\Gamma \implies A, \Delta}{\Gamma \implies A \vee B, \Delta}$ $\vee$-right^1-Gentzen (e.g., "$q \implies p \vee q$" is provable, but "$q \implies p$" is not)

Hands-On: Optimize Invertible Rules

- try to prove the formula in `ex_pigeon3.pl` with `leanseq_v1.pl`
- try to refute the formula in `ex_invalid.pl` with `leanseq_v1.pl`
- add a cut "`!`" in the axiom and in each rule after "`select1`"
- `leanseq_v2.pl`:

```
% axiom
prove(G > D) :- member(A,G), member(A,D), !.
% conjunction
prove(G > D) :- select1(A&B,G,G1), !,
                prove([A,B|G1] > D).
% ... add "!" to seven more rules
```

- try to prove the formula `ex_pigeon3.pl` with `leanseq_v2.pl`
- try to refute the formula `ex_invalid.pl` with `leanseq_v2.pl`

Build Your Own First-Order Prover

Part 2b: Implementing a First-Order Prover

Jens Otten

University of Oslo

Hands-On: "Barber Puzzle"

"There is a man in Natal who shaves all the men in Natal who do not shave themselves."

Question: "Does some man in Natal shave himself."

- formalize this puzzle in the language of (first-order) logic!
- $\exists x \forall y \, (\neg s(y,y) \rightarrow s(x,y)) \rightarrow \exists z \, s(z,z)$
- express the "barber puzzle" in Prolog syntax!
- `ex_barber.pl`:

```
fof(barber,conjecture,
    ?[X]: ![Y]: ( ~s(Y,Y) => s(X,Y) ) => ?[Z]: s(Z,Z) ).
```


Sequent Calculus – Rules for ∀-left and ∃-right

- rules for "universally quantified" variables

$$\frac{\Gamma, A[x\backslash t], \forall x\, A \implies \Delta}{\Gamma, \forall x\, A \implies \Delta}\ \forall\text{-left} \qquad \frac{\Gamma \implies \exists x\, A, A[x\backslash t], \Delta}{\Gamma \implies \exists x\, A, \Delta}\ \exists\text{-right}$$

  - $A[x\backslash t]$ is the formula A in which all free occurrences of x have been replaced by the term t; t is an arbitrary term
  - the formula $\forall x\, A$ is preserved in the premise of the rule ∀-left and formula $\exists x\, A$ is preserved in the premise of the rule ∃-right
- when applying these rules we have to make a wise choice for t
- Example: $\forall x\, p(x) \rightarrow p(a)$

$$\dfrac{\dfrac{\dfrac{}{p(b), \forall x\, p(x) \implies p(a)}\ ??}{\forall x\, p(x) \implies p(a)}\ \forall\text{-left}}{\implies \forall x\, p(x) \rightarrow p(a)}\ \rightarrow\text{-right} \qquad \dfrac{\dfrac{\dfrac{}{p(a), \forall x\, p(x) \implies p(a)}\ \text{axiom}}{\forall x\, p(x) \implies p(a)}\ \forall\text{-left}}{\implies \forall x\, p(x) \rightarrow p(a)}\ \rightarrow\text{-right}$$

Using Free Variables

- instead of guessing the right term t for the variable x, we delay this choice until an axiom is reached, i.e., we use "free variables"
- rules for "universally quantified" variables using free variables

$$\frac{\Gamma, A[x\backslash x'], \forall x\, A \implies \Delta}{\Gamma, \forall x\, A \implies \Delta}\ \forall\text{-left} \qquad \frac{\Gamma \implies \exists x\, A, A[x\backslash x'], \Delta}{\Gamma \implies \exists x\, A, \Delta}\ \exists\text{-right}$$

  - $A[x\backslash x']$ is a copy of the formula A in which all free occurrences of x have been replaced by the new variable x'
- axiom using free variables

$$\frac{}{\Gamma, A \implies A', \Delta}\ \text{axiom}$$

  - the two formulae A and A' need to "unify" under a single substitution
- Example:

$$\dfrac{\dfrac{\dfrac{}{p(x'), \forall x\, p(x) \implies p(a)}\ \text{axiom'}\ (x'{=}a \rightsquigarrow x'\backslash a)}{\forall x\, p(x) \implies p(a)}\ \forall\text{-left'}}{\implies \forall x\, p(x) \rightarrow p(a)}\ \rightarrow\text{-right}$$

Implementing Free Variables – Quantifier Rules

- rules for "universally quantified" variables using free variables

$$\frac{\Gamma_1, \forall x\, A, A[x\backslash y] \implies \Delta}{\Gamma_1, \forall x\, A \implies \Delta}\ \forall\text{-left} \qquad \frac{\Gamma \implies \exists x\, A, \Delta_1, A[x\backslash y]}{\Gamma \implies \exists x\, A, \Delta_1}\ \exists\text{-right}$$

  - $A_1{=}A[x\backslash y]$ is copy of formula A with x replaced by new variable y
- `leanseq_v3.pl`:

```
% universal quantifier
prove(G > D,FV) :- member((![X]:A),G),
                   copy_term((X:A,FV),(Y:A1,FV)),
                   prove([A1|G] > D,[Y|FV]).
% existential quantifier
prove(G > D,FV) :- member((?[X]:A),D),
                   copy_term((X:A,FV),(Y:A1,FV)),
                   prove(G > [A1|D],[Y|FV]).
```

- `copy_term(S,T)` creates copy T of Prolog term S with fresh variables
- free variables in A must not be renamed; a list of these variables `FV` is added to all rules (and axiom); the free variable `Y` is added to this list

Adding List FV of Free Variables

- the list `FV` of free variables is added as argument to the Prolog predicate `prove`, i.e., `prove(G > D,FV)` instead of `prove(G > D)`
- this argument is also added to the `prove` predicate that invokes the actual prover; at the beginning this list is empty
- `leanseq_v3.pl`:

```
prove(F) :- prove([] > [F],[]).

% conjunction
prove(G > D,FV) :- select1(A&B,G,G1), !,
                   prove([A,B|G1] > D,FV).

prove(G > D,FV) :- select1(A&B,D,D1), !,
                   prove(G > [A|D1],FV),
                   prove(G > [B|D1],FV).

% ... add argument "FV" to six more rules
```


Implementing Free Variables – Axiom

- axiom using free variables

$$\frac{}{\Gamma, A \implies B, \Delta}\ \text{axiom}$$

  - the two formulae A and B need to "unify" under a single substitution
- `leanseq_v3.pl`:

```
% axiom
prove(G > D,_) :- member(A,G), member(B,D),
                  unify_with_occurs_check(A,B).
```

- `unify_with_occurs_check(A,B)` is a built-in Prolog predicate that implements sound term unification
- the unification also ensures that `A` and `B` have the same predicate

Hands-On: Run Your Prover

- `$> swipl` % start SWI-Prolog
- `[leanseq_v3].` % load your prover
  `make.` % reload prover after changing the source code
-
```
prove( p => p ).
prove( p(f(a)) => p(f(a)) ).
prove( ![X]:p(X) => p(a) ).
prove( ![X]:p(X) => p(a) & p(b) & p(f(a,g(b,c))) ).
prove( ![X]: p(X) => ?[Y]: p(Y) ).
```
- last formula fails as Prolog uses an incomplete depth-first search
- `halt.` % exit SWI-Prolog
- the resulting prover `leanseq_v3.pl` is sound and complete for formulae with at most one universally quantified variable
- an iterative deepening proof search is necessary in order to regain completeness

Iterative Deepening

- number `I` specifies maximum number of free variables in sequents
- this number is increased step by step, i.e., if a proof with `I` ≤ 1 does not succeed, `I` is increased to 2, afterwards to 3, and so on
- this guarantees that all possible and alternative rule applications are considered during the proof search and yields completeness
- a loop is implemented around the predicate that invokes the actual prover; it also prints the number `I` of the current iteration
- `leanseq_v4.pl`:

```
prove(F) :- prove(F,1).
prove(F,I) :- print(iteration:I), nl,
              prove([] > [F],[],I).
prove(F,I) :- I1 is I+1, prove(F,I1).
```


Adding Free Variable Limit

- the maximum limit `I` of free variables in a sequent is added as argument to the predicate `prove`, i.e., `prove(G > D,FV,I)` instead of `prove(G > D,FV)`
- a check if this limit is reached is added to the quantifier rules introducing free variables; `length(L,N)` returns size `N` of list `L`
- `leanseq_v4.pl`:

```
% axiom
prove(G > D,_,_) :- member(A,G), member(B,D),
                    unify_with_occurs_check(A,B).
% universal quantifier
prove(G > D,FV,I) :- member((![X]:A),G),
                     \+ length(FV,I),
                     copy_term((X:A,FV),(Y:A1,FV)),
                     prove([A1|G] > D,[Y|FV],I).
% the same line is added to the existential quantifier rule
% the argument "I" is added to all eight propositional rules
```


Hands-On: Run Your Prover

- `$> swipl` % start SWI-Prolog
- `[leanseq_v4].` % load your prover
  `make.` % reload prover after changing the source code
-
```
prove( p(a) => p(a) ).
prove( ![X]:p(X) => p(a) & p(b) ).
prove( ![X]: p(X) => ?[Y]: p(Y) ).
```
- obeserve that the proof search for the last formula succeeds
- `halt.` % exit SWI-Prolog
- the resulting prover `leanseq_v4.pl` is sound and complete for formulae that contain only universally quantified variables, i.e., no Eigenvariables are allowed

Sequent Calculus – Rules for $\exists$-right and $\forall$-left

- rules for "existentially quantified" variables (Eigenvariables)

$$\frac{\Gamma, A[x\backslash a] \implies \Delta}{\Gamma, \exists x\, A \implies \Delta}\ \exists\text{-left}^* \qquad \frac{\Gamma \implies A[x\backslash a], \Delta}{\Gamma \implies \forall x\, A, \Delta}\ \forall\text{-right}^*$$

  - Eigenvariable condition for these rules: constant a must not occur in the conclusion, i.e., in Γ, Δ, or A
- when applying these rules we have to respect the Eigenvariable condition; usually Eigenvariable rules have to be applied first
- Example: $\forall x\, p(x) \rightarrow \forall y\, p(y)$

$$\dfrac{\dfrac{\dfrac{??}{p(a), \forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-right}^*}{\forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-left}}{\implies \forall x\, p(x) \rightarrow \forall y\, p(y)}\ \rightarrow\text{-right} \qquad \dfrac{\dfrac{\dfrac{}{p(a), \forall x\, p(x) \implies p(a)}\ \text{axiom}}{\forall x\, p(x) \implies p(a)}\ \forall\text{-left}}{\forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-right}^* \Big/ \implies \forall x\, p(x) \rightarrow \forall y\, p(y)\ \rightarrow\text{-right}$$

- using free variables, this condition has to be checked during unification

Using Dynamic Skolemization

- instead of a constants a, we replace the variable x by a Skolem term $f_{sk}(x_1, ..., x_n)$ in which f_{sk} is a new function symbol and $x_1, ..., x_n$ are the free variables in the sequent of the conclusion
- this yields a mechanism that fails during unification if the Eigenvariable is assigned to one of its free variables $x_1, ..., x_n$
- rules for Eigenvariables using Skolemization

$$\frac{\Gamma, A[x\backslash t_{sk}] \implies \Delta}{\Gamma, \exists x\, A \implies \Delta}\ \exists\text{-left}^* \qquad \frac{\Gamma \implies A[x\backslash t_{sk}], \Delta}{\Gamma \implies \forall x\, A, \Delta}\ \forall\text{-right}^*$$

  - where t_{sk} is the Skolem term $f_{sk}(x_1, ..., x_n)$ as described above
- Example: $\forall x\, p(x) \implies \forall y\, p(y)$

$$\dfrac{\dfrac{\dfrac{x' = f_{sk1}(x') \rightsquigarrow \text{fails}}{p(x'), \forall x\, p(x) \implies p(f_{sk1}(x'))}\ \text{axiom(?)}}{p(x'), \forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-right}^*}{\forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-left} \qquad \dfrac{\dfrac{\dfrac{x' = f_{sk1}() \rightsquigarrow \text{succeeds}}{p(x'), \forall x\, p(x) \implies p(f_{sk1}())}\ \text{axiom}}{\forall x\, p(x) \implies p(f_{sk1}())}\ \forall\text{-left}}{\forall x\, p(x) \implies \forall y\, p(y)}\ \forall\text{-right}^*$$

Implementing Skolemization – Quantifier Rules

- rules for Eigenvariables using Skolemization

$$\frac{\Gamma_1, A[x\backslash t_{sk}] \implies \Delta}{\Gamma_1, \exists x\, A \implies \Delta}\ \exists\text{-left}^* \qquad \frac{\Gamma \implies A[x\backslash t_{sk}], \Delta_1}{\Gamma \implies \forall x\, A, \Delta_1}\ \forall\text{-right}^*$$

  where t_{sk} is the Skolem term $f_{sk}(x_1, ..., x_n)$
- `leanseq_v5.pl`:

```
% universal quantifier
prove(G > D,FV,I,J,K) :- select1((![X]:A),D,D1), !,
                         copy_term((X:A,FV),(f_sk(J,FV):A1,FV)),
                         J1 is J+1,
                         prove(G > [A1|D1],FV,I,J1,K).

% existential quantifier
prove(G > D,FV,I,J,K) :- select1((?[X]:A),G,G1), !,
                         copy_term((X:A,FV),(f_sk(J,FV):A1,FV)),
                         J1 is J+1,
                         prove([A1|G1] > D,FV,I,J1,K).
```

- counter for Skolem terms is added that is increased if rules are applied

Adding a Counter for Skolem Terms

- the counters `J` and `K` are added as argument to the Prolog predicate `prove`, i.e., `prove(G > D,FV,I,J,K)` instead of `prove(G > D,FV,I)`
- these arguments are also added to the `prove` predicate that invokes the actual prover; at the start the counter is set to 1
- `leanseq_v5.pl`:

```
prove(F,I) :-  print(iteration:I), nl,
               prove([] > [F],[],I,1,_).

% conjunction
prove(G > D,FV,I,J,K) :- select1(A&B,G,G1), !,
                         prove([A,B|G1] > D,FV,I,J,K).

prove(G > D,FV,I,J,K) :- select1(A&B,D,D1), !,
                         prove(G > [A|D1],FV,I,J,J1),
                         prove(G > [B|D1],FV,I,J1,K).

% ... add arguments "J" and "K" to six more rules
```


Implementing Skolemization – Updating the Axiom

- in order to keep soundness, the axiom can only be applied to atomic formulae and not to arbitrary formulae anymore
- counter example is the invalid formula $\forall x\, p(a) \rightarrow \forall y\, p(y)$
- `leanseq_v5.pl`:

```
% axiom
prove(G > D,_,_,J,J) :- member(A,G),
                        A\=(_&_), A\=(_|_), A\=(_=>_),
                        A\=(~_), A\=(!_), A\=(?_),
                        member(B,D),
                        unify_with_occurs_check(A,B).
```

- the resulting prover `leanseq_v5.pl` is sound and complete for (classical) first-order logic

Hands-On: Run Your Prover

- `$> swipl` % start SWI-Prolog
- `[leanseq_v5].` % load your prover
  `make.` % reload prover after changing the source code
- ```
  prove( p(a) => p(a) ).
  prove( ![X]:p(X) => p(a) & p(b) ).
  prove( ![X]: p(X) => ?[Y]: p(Y) ).
  prove( ?[X]: p(X) => ![Y]: p(Y) ).
  prove( ![X]: ?[Y]: p(X,Y) => ?[U]: ![V]: p(V,U) ).
  ```
- `[ex_barber].` % load the barber puzzle
  `fof(barber,_,F).` % check the loaded formula
  `fof(barber,_,F), prove(F).` % solve puzzle
- `halt.` % exit SWI-Prolog
  ```