How to Build an Automated Theorem Prover

Part 1a: ATP and Proof Calculi

Jens Otten

University of Oslo

Our Plan for Today

Part 1a: ATP and Proof Calculi

Part 1b: Prolog

Part 2a: Implementing a Propositional Prover

Part 2b: Implementing a First-Order Prover

Part 3a: A Tableau Prover

Part 3b: A Connection Prover

Acknowledgement: The author would like to thank Andrei Popescu for the invitation and Pascal Fontaine for coming up with the idea for this tutorial.

Language of Logic

- Propositional logic

(a') Socrates is a man	*man(Socrates)*
(a) Plato is a man	*man(Plato)*
(b) if Plato is a man, then Plato is mortal	$man(Plato) \rightarrow mortal(Plato)$
(c) Plato is mortal	*mortal(Plato)*

- First-order logic

(a') *man(Socrates)*
(a) *man(Plato)*
(b') $\forall x\ (\ man(x) \rightarrow mortal(x)\ )$
(c') $mortal(Plato) \wedge mortal(Socrates)$

- First-order (predicate) logic: predicates, $\rightarrow$, $\wedge$, $\vee$, $\neg$, $\forall x$, $\exists x$

First-Order Logic – Syntax

- Terms (s, t, u, v) are inductively defined as follows:
  1. Every variable $(x, y, z, ...)$ and every constant $(a, b, c, ...)$ is a term.
  2. Let f (also $g, h, ...$) be a function symbol and $t_1, ..., t_n$ be terms, then $f(t_1, ..., t_n)$ is also a term.
- Atomic formulae (P) are defined as follows:
  1. Every predicate symbol (p, q, r) is an atomic formula.
  2. If P is a predicate symbol and $t_1, ..., t_n$ are terms, then $P(t_1, ..., t_n)$ is an atomic formula.
- First-order formulae (A, B, C, F, G) are defined as follows:
  1. Every atomic formula P is a formula.
  2. If A and B are formulae and x is a variable, then $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $\forall x\, A$, and $\exists x\, A$ are formulae.

Logical Consequence

- Given: Finite set of formulae F_1, F_2, ..., F_n ("axioms"), formula F ("conjecture")
- Question: Is F a logical consequence of F_1, F_2, ..., F_n?
- Answer: Yes, iff (if and only if) $F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow F$ is valid

Deduction Theorem:

- Logical consequence can be reduced to validity

Logical Validity

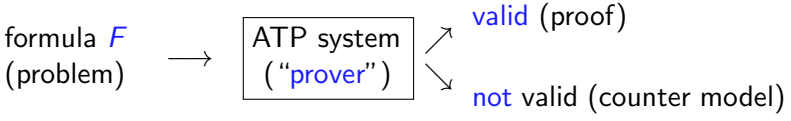
- Logical validity: A formula F is valid iff F is true for all possible interpretations of its predicate, constant and function symbols
- Examples of valid formulae:
  $man(Plato) \wedge (man(Plato) \rightarrow mortal(Plato)) \rightarrow mortal(Plato)$
  $man(Plato) \wedge \forall x(man(x) \rightarrow mortal(x)) \rightarrow mortal(Plato)$
  $p \vee \neg p$ (not in intuitionistic logic!),
  $(((\exists x\, q(x) \vee \neg q(c)) \rightarrow p) \wedge (p \rightarrow (\exists y\, q(y) \wedge r))) \rightarrow (p \wedge r)$
- Tautology, i.e. deciding validity in propositional logic is co-NP-complete (F ist valid iff $\neg F$ ist not satisfiable)
- Validity in first-order logic is only semi-decidable

Automated Theorem Proving

Automated Theorem Proving (ATP) is a core research area in the field of Artificial Intelligence.

Goal: automating logical reasoning in (non-)classical logics

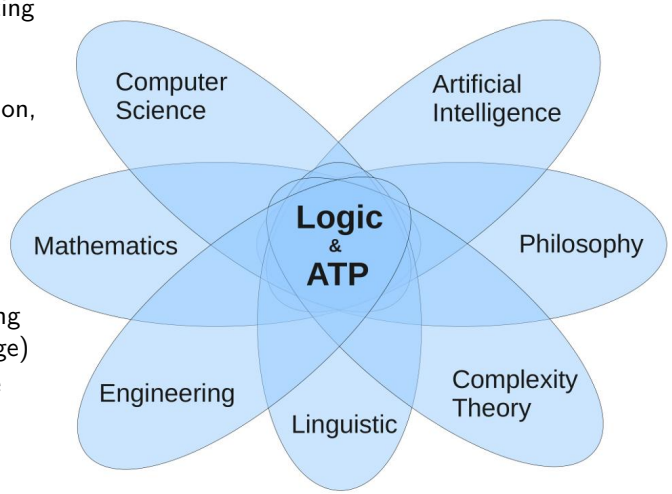
- is a given conjecture a logical consequence of a set of axioms?
- is a given formula valid with respect to a specific logic?



- main challenge: complexity, i.e. efficient proof search

Logic and ATP

- Philosophy (formalizing truth and reasoning)
- Computer Science (modelling, verification, logic programming)
- Mathematics (proof theory)
- Engineering (modelling ICs)
- Linguistic (formalizing semantics of language)
- Artificial Intelligence (formalizing and reasoning)
- Complexity Theory (NP-completeness)



What is a Calculus?

- "a particular method or system of calculation or reasoning"
- formal calculus := language $\{w, w_1, w_2, ...\}$ + axioms + rules

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)

- a derivation of w is a tree
  - whose nodes are axioms or rules of the calculus and
  - the premises of each inner node are conclusions of its parent nodes
- a proof of w is a derivation of w whose leaves are axioms

Proof Calculi

- language: first-order formulae
- formula F is valid $\Leftrightarrow$ there is a proof for F in a proof calculus

Some popular proof calculi:

- Natural Deduction [Gentzen 1935]
  (classical and intuitionistic logic, NK and NJ)
- Sequent Calculus [Gentzen 1935]
  (classical and intuitionistic logic, LK and LJ)
- Tableau Calculus [Beth 1955, Smullyan 1968]
- DPLL Calculus [Davis/Putnam 1960,Davis/Logemann/Loveland 1962]
- Resolution Calculus [Robinson 1965]
- Model Elimination [Loveland 1968] (similar to connection calculus)
- Connection Calculus [Bibel 1981]
- Instance-based Methods [Lee & Plaisted 1992]

The Sequent Calculus

A sequent has the form

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

where Γ and Δ are finite (possibly empty) multisets of formulae.

- left side of sequent is the antecedent, right side is the succedent
- $\Gamma \cup \{A\}$ or $\Delta \cup \{B\}$ are usually written as Γ, A and Δ, B, respectively
- intuitively, a sequent represents "provable from" in the sense that the formulae in Γ are assumptions for the set of formulae Δ to be proven
- a sequent $A_1, \ldots, A_n \implies B_1, \ldots, B_m$ can be interpreted as $(A_1 \wedge \ldots \wedge A_n) \rightarrow (B_1 \vee \ldots \vee B_m)$

There are rules for eliminating connectives and quantifiers in sequents.

- a proof of formula A is a proof of the sequent $\implies A$
- a formula A is provable, written $\vdash A$, iff there is a proof for A

Sequent Calculus – Axiom

- the only axiom

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

Sequent Calculus – Rules for $\wedge$ and $\vee$

- rules for $\wedge$ (conjunction)

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

- rules for $\vee$ (disjunction)

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

Sequent Calculus – Rules for $\rightarrow$ and $\neg$

- rules for $\rightarrow$ (implication)

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

- rules for $\neg$ (negation)

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

Sequent Calculus – Examples

Example: $p \wedge (p \rightarrow q) \rightarrow q$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \implies p, q}{}\ \text{axiom} \quad \dfrac{p, q \implies q}{}\ \text{axiom}}{p, p \rightarrow q \implies q}\ \rightarrow\text{-left}}{p \wedge (p \rightarrow q) \implies q}\ \wedge\text{-left}}{\implies p \wedge (p \rightarrow q) \rightarrow q}\ \rightarrow\text{-right}$$

Example: $(\neg p \vee q) \rightarrow (p \rightarrow q)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p \implies p, q}{}\ \text{axiom}}{\neg p, p \implies q}\ \neg\text{-left} \quad \dfrac{q, p \implies q}{}\ \text{axiom}}{\neg p \vee q, p \implies q}\ \vee\text{-left}}{\neg p \vee q \implies p \rightarrow q}\ \rightarrow\text{-right}}{\implies (\neg p \vee q) \rightarrow (p \rightarrow q)}\ \rightarrow\text{-right}$$

Sequent Calculus – Rules for $\forall$ and $\exists$

- rules for $\forall$ (universal quantifier)

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

  - t is an arbitrary term
  - Eigenvariable condition for the rule $\forall$-right*: a must not occur in the conclusion, i.e. in Γ, Δ, or A
  - the formula $\forall x\, A$ is preserved in the premise of the rule $\forall$-left

- rules for $\exists$ (existential quantifier)

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

  - t is an arbitrary term
  - Eigenvariable condition for the rule $\exists$-left*: a must not occur in the conclusion, i.e. in Γ, Δ, or A
  - the formula $\exists x\, A$ is preserved in the premise of the rule $\exists$-right

Sequent Calculus – First-Order Examples

Example: $\forall x\, p(x) \to \exists x\, p(x)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(c), \forall x\, p(x) \implies p(c), \exists x\, p(x)}\text{ axiom}}{p(c), \forall x\, p(x) \implies \exists x\, p(x)}\exists\text{-right}}{\forall x\, p(x) \implies \exists x\, p(x)}\forall\text{-left}}{\implies \forall x\, p(x) \to \exists x\, p(x)}\to\text{-right}$$

Example: $\exists x\, p(x) \to p(a)$

$$\dfrac{\dfrac{}{\exists x\, p(x) \implies p(a)}\exists\text{-left}^*}{\implies \exists x\, p(x) \to p(a)}\to\text{-right}$$

rule $\exists$-left*with $p(x)[x\backslash a]$ cannot be applied as a occurs in the premise (Eigenvariable condition!)

Sequent Calculus – Soundness and Completeness

Theorem (Soundness and Completeness of LK)

The first-order sequent calculus LK is sound and complete, i.e.

- *if A is provable in LK, then A is valid (if $\vdash A$ then $\models A$)*
- *if A is valid, then A is provable in LK (if $\models A$ then $\vdash A$)*

Proof.
See [Gentzen 1935]. □

How to Build an Automated Theorem Prover

Part 1b: Prolog

Jens Otten

University of Oslo

Prolog

- Prolog = Programming in Logic
- declarative programming language: specify the problem and let the computer solve it
- invented in 1970th by A. Colmerauer, R. Kowalski, P. Roussel.
- computation is searching for a proof, output is assignment of variables in the proof
- algorithm = logic + control [Kowalski 1979]
- logic: Prolog program is specified by set of Horn clauses in a restricted language of first-order logic
- control: Prolog program is "executed" by the Prolog interpreter/"prover"

Prolog – An Example

- An example in Prolog (file `family.pl`)

```
male(thomas).                          % these are facts
male(rolf).
female(anna).
female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).

father(X,Y) :- parent(X,Y), male(X).   % these are rules
mother(X,Y) :- parent(X,Y), female(X).

grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

- start Prolog and type `'[family].'` to load the program
- `Ctrl-C` stops Prolog; `'halt.'` exits Prolog

Prolog Queries – Examples

- `?- parent(maria,anna).`
  `true.`
  `?- parent(anna,maria).`
  `false.`
- `?- parent(X,anna).`
  `X = thomas  <press ';' for more solutions>`
  `X = maria  <press ';' for more solutions>`
  `false.`
- `?- father(X,Y).`
  `X = thomas,`
  `Y = anna  <press ';' for more solutions>`
  `X = rolf,`
  `Y = maria.`
- `?- grandfather(rolf,Y).`
  `Y = anna.`

Prolog – Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)
- variables: start with upper case letter or the underscore `'_'`
  (e.g. `X`, `Y`, `Number`, `List`, `_ABC`; `'_'` is anonymous variable)
- structures: ⟨*constant*⟩ or ⟨*constant*⟩(*Term*1, ..., *TermN*)
  (e.g. `parent(maria,anna)`)

Predicates ⟨*predicate*⟩:

- ⟨*constant*⟩ or ⟨*constant*⟩(*Term*1, ..., *TermN*)
  (e.g. `thomas`, `parent(maria,anna)`)

Prolog – Facts, Rules and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule. The user can query the Prolog program/database.

Facts:

- ⟨*predicate*⟩. % observe the '.' at the end (this is a comment)
  e.g. `male(rolf).` or `parent(maria,anna).`

Rules:

- ⟨*predicate*⟩ `:-` ⟨*predicate*1⟩, ... , ⟨*predicateN*⟩.
  e.g. `father(X,Y) :- parent(X,Y), male(X).`
- rules have the form *Head* `:-` *Body*.
- `':-'` can be read as '←'; the comma `','` in the body can be read as '∧'

Query:

- ⟨*predicate*1⟩, ... , ⟨*predicateN*⟩.
  e.g. `parent(maria,anna).` or `grandfather(rolf,Y).`

Operational Semantics

- Prolog tries to prove the query using the facts and rules in its database
- it starts trying to fulfil/solve the predicates one after the other
- if an appropriate fact matches, then the predicate/goal succeeds
- if the head of a rule matches, then Prolog continues by trying to fulfil the predicates of the rule's body
- the database is searched top to bottom
- if more than one fact or head of a rule matches, then alternative options are considered if the search fails (via backtracking)

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                               -> parent(maria, anna)
                  male(maria) -> fail
                               -> parent(rolf, maria)
                  male(rolf)
   parent(maria,anna)
grandfather(rolf,anna) succeeds
X = rolf.
```

- variables are instantiated ("bound") during the unification of terms

Logical Semantics

The semantics of a program is specified by the following formula F.

`fact_1.`	`( fact_1`
`...`	$\wedge$ `...`
`fact_n.`	$\wedge$ `fact_n`
`head_1 :- body_1.`	$\wedge$ `head_1` $\leftarrow$ `body_1`
`...`	$\wedge$ `...`
`head_m :- body_m.`	$\wedge$ `head_m` $\leftarrow$ `body_m )`
`?- query.`	$\rightarrow$ `query`

The query succeeds iff the Prolog program terminates and F is valid.

- variables are quantified in the following way:

  $\forall X1, ..., Xn \ (\exists Y1, ... Yn$ `body_i` $\rightarrow$ `head_i`$)$

  for all variables $X1, ..., Xn$ occurring in `head_i` and all variables $Y1, ... Yn$ occurring in `body_i`
- inference engine is a theorem prover based on SLD resolution (only horn clauses, depth-first search (incomplete!), no occurs-check (unsound!))

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

-
```
?- [H|T]=[a,b,c,d].
H = a,
T = [b, c, d].

?- [H1,H2|T]=[a,b,c,d].
H1 = a,
H2 = b,
T = [c, d].
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
X = a ;
X = b .

?- append([a,b],[c],Z).
Z = [a, b, c].
?- append(X,Y,[a,b,c]).
X = [], Y = [a, b, c] ;
X = [a], Y = [b, c] ;
X = [a, b], Y = [c] ;
X = [a, b, c], Y = [] .
```


Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- arithmetic operators '=', '<', '>', '>=', '=<' are interpreted predicates
- $0! = 1$, $n! = n * (n-1)!$ if $n>0$:

```
factorial(0,1).
factorial(N,I) :- N>0, N1 is N-1,
                  factorial(N1,I1), I is N*I1.

?- factorial(5,I).
I = 120.
```


Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '\+ predicate' succeeds iff 'predicate' fails
-
```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).

?- female(kristine).
false.

?- \+ female(kristine).
true.

?- \+ parent(rolf,thomas).
true.
```


Cut, Disjunction and If-Then-Else

- the cut '!' is used to control Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when trying to fulfil the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
-
```
factorial(0,1) :- !.
factorial(I,N) :- I1 is I-1, factorial(I1,N1), N is I*N1.
```

- `predicate :- predicate1 ; predicate2`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1`/`predicate2` when re-fulfilling `predicate`

```
grandparent(X,Y) :- grandfather(X,Y) ; grandmother(X,Y).
```

- `Cond -> Goal1 ; Goal2`
  succeeds iff Cond succeeds and Goal1 succeeds or Cond fails and Goal2 succeeds; no backtracking within Cond ("implicit cut")

```
f(I,N) :- I=0 -> N=1 ; I1 is I-1, f(I1,N1), N is I*N1.
```