Computational Logic

Automated Theorem Proving

Lecture 1

Cezary Kaliszyk

May 19, 2020

Administration

Teacher

- Cezary Kaliszyk
- Consultation hours: This semester by email only

VU

- Virtual Lectures: Notes for self-study in the given course slots
- Virtual Exercises: Tasks to be performed at the end of the lecture
- You need to submit the exercises to me by email, and I will comment about your solutions only if there is need.
- For simplicity, the deadlines for the exercises are next lecture dates.

Grading

- (Remote) open-book exam
- Practical assignments
- Final grade: average of the two

Covered Topics

Basics

Why ATPs?

Tableaux in practice

leanCoP and variants

Resolution

Orderings

Advanced

Paramodulation

Superposition

ATP in intuitionistic logic

ATP in higher-order logic

Outline

Today

- What is automated theorem proving
- Common uses and differences to other systems
- Tableaux in practice

Automated Reasoning

Field of research since the fifties

- Computer used to reason in a logic
- Traditionally part of artificial intelligence
  - (not machine learning)
- Applications: program verification, mathematical deduction, ...
- Theorem proving logics, precision, automation, ... very varied.
- In this course we look at automated theorem proving, which is a subpart of this field

Automated Theorem Proving: Main Question

Given

A set of axioms and a conjecture in a formal language
(e.g. first-order logic)

$$\{A_1, ..., A_n\}, C$$

Answer

Do the axioms logically imply the conjecture?

$$A_1, ..., A_n \models C$$

Automated Theorem Proving: Main Question

Given

A set of axioms and a conjecture in a formal language
(e.g. first-order logic)

$$\{A_1, ..., A_n\}, C$$

Answer

Do the axioms logically imply the conjecture?

$$A_1, ..., A_n \models C$$

Notes

- Only in very simple logics this is decidable, so the ATP will often not terminate
- Typically the system will try to prove $A_1, ..., A_n \models C$, and soundness or even completeness will be shown meta-theoretically.

Why is this interesting? (1/2)

Some higher math is slowly becoming hard for humans

Problems in some domains of algebra: quasigroup theory and loop theory

Robbins Conjecture

- Open problem for many years: Mathematicians really tried
- Simple statement: Given an associative-commutative operator in an algebra is it true that:

$$n(n(x)+y)+n(n(x)+n(y))=x \iff n(n(x+y)+n(x+n(y)))=x$$

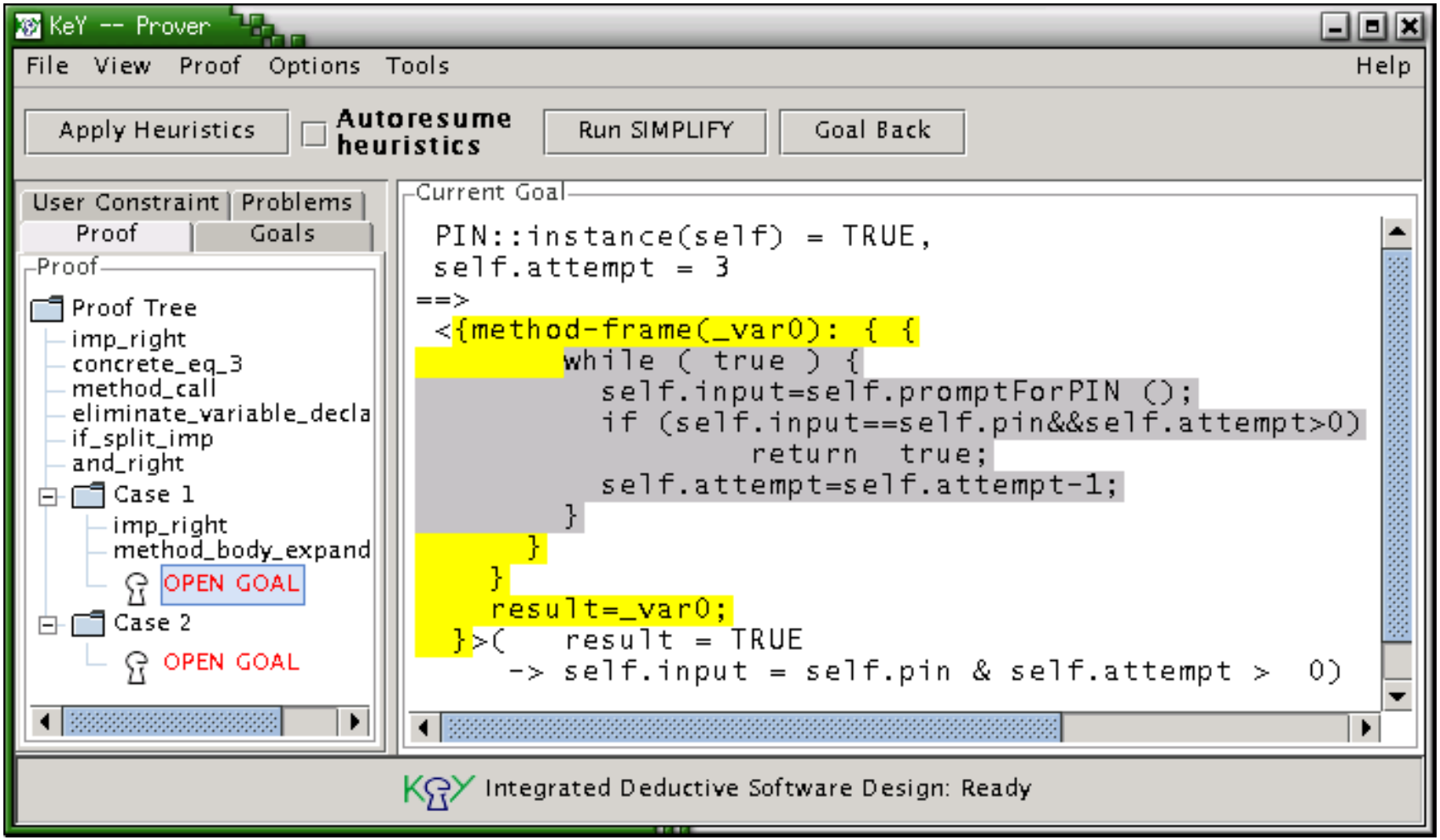
- Computer program (McCune's EQP) proved the fact in 1996.

Note

In mathematics the use of automated theorem proving is still rather limited, with many more users of computer algebra or related systems.

Why is this interesting (2/2)

Verification of programs or security protocols is often translated to many small ATP obligations



.

[Beckert]

Most work in classical first-order logic

Rationale for this

- Compromise between expressivity and automatizability

Expressive

- Any computable problem can be encoded, many of them naturally
- Some complex logics can be well-translated to FOL

Automation is possible

- Sound and complete calculi exist
- Reasonably efficient search

History of reasoning in FOL

- Resolution [*Robbins'65*]
- Model Elimination [*Loveland'68*]
- Paramodulation [*Robinson&Wos'69*]
- Completion [*Knuth&Bendix'70*]
- Superposition [*Bachmaier&Ganzinger'90*]
- leanTAP [*Beckert&Posegga'95*]
- Vampire [*Voronkov'95*]
- First CASC Competition [*Sutcliffe&Suttner'96*]
- Eprover [*Schulz'99*]
- Efficient EUF in SMT [*Nieuwenhuis'06*]
- SAT in FOL [*Voronkov'14,Schulz'19*]

Efficiency of Tableaux

Until the late nineties, tableaux based systems were dominating competitions. It will only be thanks to the handling of equality (presented in the second part of this course) that we will see an emergence resolution based systems.

We have seen Tableaux in the CL lecture. However there an abstract setting was presented without the focus on efficiency. First (reasonably) efficient, but also very concise Tableaux systems were constructed in Prolog. Later a number of improvements were added in the Setheo system. We will only look at the Prolog systems in this course.

leanTAP presentation mostly follows Otten.

Note: Today we will look at these in Prolog notation, which you may be unfamiliar with, next time we will switch to functional programming.

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 (using 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)
```

Tableau Calculus (should be known from CL course)

The (analytic/semantic) tableau calculus for formulae in negation normal form consists of α-, β-, and γ-rules (Smullyan '68).

- α-rule

$$\frac{F \wedge G}{\begin{array}{c} F \\ G \end{array}} \wedge$$

β-rule

$$\frac{F \vee G}{\begin{array}{cc} / & \backslash \\ F & G \end{array}} \vee$$

γ-rule

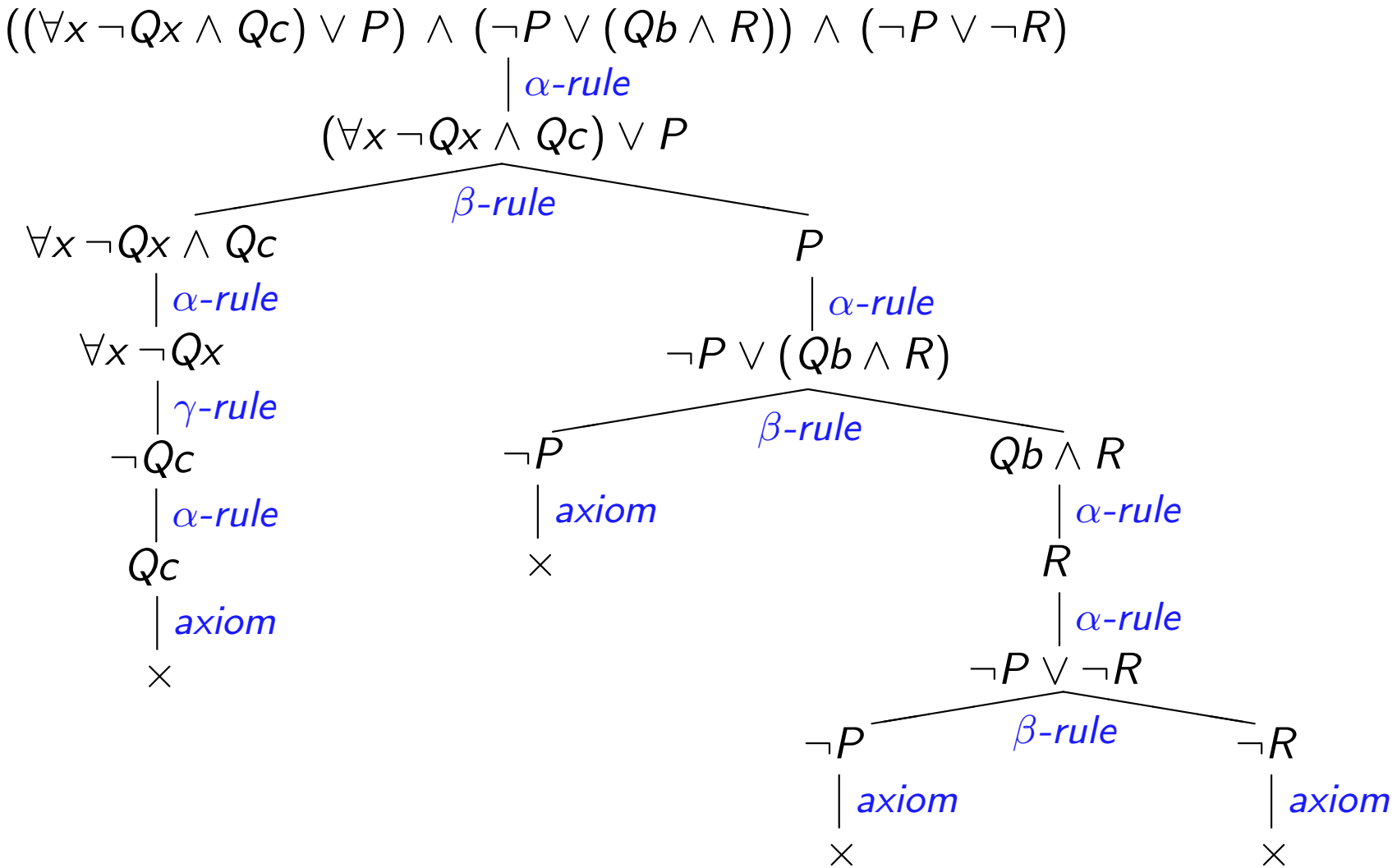
$$\frac{\forall x\, F}{F[x \backslash t]} \forall$$

- branch is closed iff it contains a set of complementary literals $\{A, \neg A\}$
- a derivation in the tableau calculus is a proof iff all branches are closed

Theorem: F is valid iff there is a proof for $\neg F$ in the tableau calculus.

- proof of Theorem: ...
- tableau calculus is confluent, i.e. no backtracking (over rules) necessary (but different terms t can be assigned to variable x in γ-rule)

Tableau Calculus – Example



What is Prolog and Why Prolog?

Prolog is a programming language that introduces "logic-like" foundations. Variables are first-order terms. Assignment of variables is performed using unification.

Isn't there some problem? One might think that:

> *We first use logic to do a programming language and then use this language to do logic*

This is partially correct. Only terms and their basic operations are done. Logical formulas that use these terms and all the reasoning done with them still need to be programmed. In particular we need to figure out a way to represent conjunctions, disjunctions, implications, ... and figure out any ways we will want to interact with them.

Finally, we need to add the algorithm for the automated reasoning on top of this.

Prolog-mini primer (1/2)

This is only used to give a minimal understanding to the next slide. We will not use Prolog in the rest of the course and I just want to show that a minimal automated prover is a rather simple program. It will be the more complex tasks that we will focus on more and we will use lower level programming for these.

A Prolog program consists of rules, for example "to do a do b and do c" could be written with three predicates as:

```
1       a :- b, c
```

Prolog also has variables, and these are implicitly universally quantified, that is "do(X)" means "do anything". If you are powerful you can do anything.

```
1       do(X) :- powerful
```

Lists starting with A and continuing with B will be represented as "[A|B]"

Prolog-mini primer (2/2)

Prolog programs can have multiple rules for a given task, e.g. if a can be obtained both from a b or from a c we could write:

```
1       a :- b
2       a :- c
```

The symbol ! is used for efficiency: This only means "You are on the right path, do not try the other rules to get this".

The symbol _ is used for variables that can be ignored: It is still assigned but will not be used afterwards.

Finally negation and implication are written as \+ and ->.

This should allow you to roughly read the next slide. The minimal Prolog prover is only given as a curiosity. We will look at the same tasks trying to do them more efficiently and functional style next time.

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

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 tableau calculus with free variables
- but performance on more difficult problems rather poor

Tableau Prover leanTaP

```
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 tableau calculus with free variables
- source code size of minimal version only 360 bytes
- requires (only) negation normal form
- `prove(Fml,[],[],[],VarLim)` succeeds iff there is a tableau for the formula `Fml` with at most `VarLim` free variables on each branch

leanTaP implementation

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

The following code implements the tableau calculus in Prolog.

- α-rule

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

- β-rule

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

leanTaP implementation

- γ-rule

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

- axiom (closes a branch)

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

- (select next unexpanded formula on branch)

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

Unification with Occurs-check

- `unify(A,B)`/`unify1(A,B)` succeed iff terms/lists `A` and `B` are unifiable

```
unify(A,B) :- unify1([A],[B]).
```

- lists are empty

```
unify1([],[]).
```

- if `A` and `B` are identical terms (e.g. identical variables)

```
unify1([A|A1],[B|B1]) :- A==B, !, unify1(A1,B1).
```

- if `A`/`B` are variables and `A`/`B` do not occur in `B`/`A`: assign `B`/`A` to `A`/`B`

```
unify1([A|A1],[B|B1]):-var(A),!,not_in(A,[B]),A=B,unify1(A1,B1).
unify1([A|A1],[B|B1]):-var(B),!,not_in(B,[A]),A=B,unify1(A1,B1).
```

- otherwise, if `A`$= f(S_1, ..., S_n)$ and `B`$= f(T_1, ..., T_n)$, unify S_i and T_i

```
unify1([A|A1],[B|B1]):-A=..[F|ArgA],B=..[F|ArgB],length(ArgA,N),
     length(ArgB,N),unify1(ArgA,ArgB),unify1(A1,B1).
```

leanTaP comments

Prolog is a reasonably efficient high-level language, but we will see how doing things more directly can be better

Working in CNF will be much more efficient

But most importantly a better search strategy and strategy combinations will allow getting most of Tableaux

Additional Literature (not required)

Summary

This Lecture

- What is automated theorem proving
- Common uses and differences to other systems
- First Tableaux

Next

- Full details of Tableaux in functional style
- leanCoP
- Optimizations possible in lean tableaux
- Variants for other logics

Exercises

To be submitted before May 26

- Install Eprover `http://eprover.org/`
- Understand the TPTP format
- State the Robbins conjecture in TPTP
- Run Eprover on the problem

Send the input problem, options, and E-prover output to `cezary.kaliszyk@uibk.ac.at`