Automatic Theorem Proving

A Very Brief Introduction
Dr. Lutz Hamel
Computer Science & Statistics
Definition

- Automated theorem proving (also known as ATP or automated deduction) is a subfield of automated reasoning and mathematical logic dealing with proving mathematical theorems by computer programs.

The Dream

source: http://www.cs.utexas.edu/users/moore/
Good News

- First-order logic together with set theory is expressive enough to serve as a foundation for mathematics
- Frege, Whitehead, Russel
For all models $M$ of some theory $T$, $T \vdash s$ iff $M \models s$
Some Bad News

- First-order logic is semi-decidable – Church/Turing
  - Given some decision procedure P:
    - P will accept and return a proof for some sentence s if s is valid.
    - P can reject or loop forever if s is not valid.

⇒ The first blow to our dream!
More Bad News

- Any decision procedure $P$ given some valid sentence $s$ runs at best in $NP$ time.
  - That is, the time it takes to run $P$ grows exponentially with the complexity of the sentence $s$.

$\Rightarrow$ The second blow to our dream!
Problem

- If an ATP runs a long time you don’t know if the cause of this is the undecidability problem or the NP problem.
Goedel’s Incompleteness Theorem

- Even though first-order logic is sound and complete there are some domains that are not finitely axiomatizable – that is there are no finite theories that describe this domain,
  - e.g. arithmetic
- This implies that any finite representation $A$ of some infinite theory $T$ such as arithmetic is incomplete,

For all models $M$ of some theory $T$ and $A \subseteq T$ is finite, $A \vdash s$ implies $M \models s$
Perhaps Some More Bad News

- Even if we accept the previous issues and continue to press on...
- …the proofs that some decision procedure is likely to construct are completely unstructured

source: prover9 proof archive
Some Successes

- Perhaps the most famous success in fully automatic theorem proving is the proof of the *Robbins Conjecture*:
  - A problem first posed by E.V. Huntington in 1933 and then refined by Herbert Robbins:
    
    For all elements $a$, $b$, and $c$:
    
    1. **Associativity**: $a \lor (b \lor c) = (a \lor b) \lor c$
    2. **Commutativity**: $a \lor b = b \lor a$
    3. **Robbins equation**: $\neg (\neg (a \lor b) \lor \neg (a \lor \neg b)) = a$

- Are all Robbins algebras Boolean?
- Yes! – proved by William McCune with the theorem prover EQP in 1996 – it took 172 hrs ≈ 1 week

source: http://www.cs.unm.edu/~mccune/papers/robbins/
Other Fully Automatic TPs

- **E** - 

- **ACL2** - 

- **Prover9** - 

- Many others
Yet…

- After almost 50 years of research in fully automatic theorem proving the results are pretty thin…
- …perhaps a better strategy is a collaboration between proof author and automatic theorem prover.
Proof Assistants
In computer science and mathematical logic, a proof assistant or interactive theorem prover is a software tool to assist with the development of formal proofs by human-machine collaboration. This involves some sort of interactive proof editor, or other interface, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer.
Proof Assistants

- Proof assistants avoid decidability problems by relying on the human to structure the proof in such a way that only valid sentences need to be validated.
- Proof assistants avoid the NP problems because typically proofs are broken down into small steps that don’t require a lot of search in order to be validated.
- Interesting ramification: logics used in proof assistants do not have to be complete!
  - We no longer rely on the fact that the TP has to potentially show that everything that is true in the models can be proven.
  - Rather, we rely on the fact that the conclusion follows from the premises.
  - This allows us to use much more powerful logics in proof assistants that would be possible in fully automatic theorem provers.
The Mizar System

- Perhaps the oldest proof assistant – started in 1973 by Andrzej Trybulec.
- Based on first-order logic and set theory
- Very large library of existing proofs – as of 2012:
  - 1150 articles written by 241 authors
  - these contain more than 10,000 formal definitions of mathematical objects and about 52,000 theorems proved on these objects
  - some examples are: Hahn–Banach theorem, König's lemma, Brouwer fixed point theorem, Gödel's completeness theorem and Jordan curve theorem.

http://mizar.org/
A Simple Mizar Proof: √2 is irrational

```plaintext
theorem
  sqrt 2 is irrational
proof
  assume sqrt 2 is rational;
  then consider i being Integer, n being Nat such that
  W1: n<>0 and
  W2: sqrt 2=i/n and
  W3: for i1 being Integer, n1 being Nat st n1<>0 & sqrt 2=i1/n1 holds n<=n1
    by RAT_1:25;
  A5: i=sqrt 2*n by W1,XCMPLX_1:88,W2;
  C: sqrt 2>=0 & n>0 by W1,NAT_1:19,SQUARE_1:93;
    then i>=0 by A5,REAL_2:121;
    then reconsider m = i as Nat by INT_1:16;
  A6: m*m = n*n*(sqrt 2*sqrt 2) by A5
    .= n*n*(sqrt 2)^2 by SQUARE_1:def 3
    .= 2*(n*n) by SQUARE_1:def 4;
    then 2 divides m*m by NAT_1:def 3;
    then 2 divides m by INT_1:44,NEWTON:98;
    then consider m1 being Nat such that
  W4: m=2*m1 by NAT_1:16;
  m1*m1*2*2 = m1*(m1*2)*2
    .= 2*(n*n) by W4,A6,XCMPLX_1:92;
    then 2 divides m1*m1 by XCMPLX_1:15;
    then 2 divides n*n by NAT_1:def 3;
    then 2 divides n by INT_2:44,NEWTON:98;
    then consider n1 being Nat such that
  W5: n=2*n1 by NAT_1:16;
A10: m1/n1 = sqrt 2 by W4,W5,XCMPLX_1:92,W2;
A11: n1>0 by W5,C,REAL_2:123;
    then 2*n1>1*n1 by REAL_2:199;
    hence contradiction by A10,W5,A11,W3;
end;
```

source: Freek Wiedijk’s book *The Seventeen Provers of the World*
The Coq System

- Started in 1984
- Implements a higher order logic: higher-order type theory
  - Implication – not complete and not decidable but sound
  - yet very expressive
- Coq is used in a large variety of domains such as formalization of mathematics, specification and verification of computer programs, etc.

source: https://coq.inria.fr
Theorem irrational_sqrt_2: irrational (sqrt 2%nat).
intros p q H0; case H.
apply (main_thm (Zabs_nat p)).
replace (Div2.double (q * q)) with (2 * (q * q));
[dtac | unfold Div2.double; ring].
case (eq_nat_dec (Zabs_nat p * Zabs_nat p) (2 * (q * q))); auto; intros H1.
case (not_nm_INR _ _ H1); (repeat rewrite mult_INR).
rewrite <- (sqrt_def (INR 2)); auto with real.
rewrite H0; auto with real.
assert (q <> 0%R :> R); auto with real.
field; auto with real; case p; simpl; intros; ring.
Qed.

main_thm =
fun n : nat =>
lt_wf_ind n
(flat n0 : nat => forall p : nat, n0 * n0 = Div2.double (p * p) -> p = 0)
(flat (n0 : nat)
  (H : forall m : nat,
   m < n0 -> forall p : nat, m * m = Div2.double (p * p) -> p = 0)
  (p : nat) (H0 : n0 * n0 = Div2.double (p * p)) =>
match Peano_dec.eq_nat_dec n0 0 with
| left H1 =>
  let H2 :=
  eq_ind_r (fun n : nat => n * n = Div2.double (p * p) -> p = 0)
  match p as n return (0 * 0 = Div2.double (n * n) -> n = 0) with
  | O => fun H2 : 0 * 0 = Div2.double (0 * 0) => H2
  | S n0 =>
    fun H2 : 0 * 0 = Div2.double (S n0 * S n0) =>
    let H3 :=
    eq_ind (0 * 0)
    (fun ee : nat =>
      match ee with
      | O => True
      | S _ => False
      end) I (Div2.double (S n0 * S n0)) H2 in
    False_ind (Div2.double (S n0 = 0) H3
    end H1 in
  H2 H0
| right H1 => ....
Isabelle

- Isabelle is a proof assistant which implements higher-order logic:
  - LCF – lambda calculus extended with logical constructs
  - incomplete, undecidable, but sound
- Isabelle is developed at University of Cambridge (Larry Paulson), Technische Universität München (Tobias Nipkow) and Université Paris-Sud (Makarius Wenzel).
- The main application is the formalization of mathematical proofs and in particular *formal verification*, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols.

source: http://isabelle.in.tum.de
Example: $\sqrt{2}$ is irrational

```
theorem sqrt2_not_rational:
  "sqrt (real 2) \notin \mathbb{Q}"
proof
  assume "sqrt (real 2) \notin \mathbb{Q}"
  then obtain m n :: nat where
    n_nonzero: "n \neq 0" and sqrt_rat: "|sqrt (real 2)| = real m / real n"
    and lowest_terms: "gcd m n = 1" ..
  from n_nonzero and sqrt_rat have "real m = |sqrt (real 2)| \cdot real n" by simp
  then have "real (m^2) = (sqrt (real 2))^2 \cdot real (n^2)" by (auto simp add: power2_eq_square)
  also have "(sqrt (real 2))^2 = real 2" by simp
  also have "... \cdot real (m^2) = real (2 \cdot n^2)" by simp
  finally have eq: "m^2 = 2 \cdot n^2" ..
  hence "2 dvd m^2" ..
  with two_is_prime have dvd_m: "2 dvd m" by (rule prime_dvd_power_two)
  then obtain k where "m = 2 \cdot k" ..
  with eq have "2 \cdot n^2 = 2^2 \cdot k^2" by (auto simp add: power2_eq_square mult_ac)
  hence "n^2 = 2 \cdot k^2" by simp
  hence "2 dvd n^2" ..
  with two_is_prime have "2 dvd n" by (rule prime_dvd_power_two)
  with dvd_m have "2 dvd gcd m n" by (rule gcd_greatest)
  with lowest_terms have "2 dvd 1" by simp
  thus False by arith
qed
```
Observations

● Pros:
  ● Powerful reasoning mechanisms – deduction, induction, tactics, etc
  ● Expressive proof languages

● Cons:
  ● steep learning curve for the systems
  ● the complicated proof languages represent an adoption hurdle
Prolog as a Proof Assistant

- I am interested in ATP coming from a formal semantics for programming languages angle:
  - build programming language models
  - reason about these models
Prolog as a Proof Assistant

I needed the following:

- a language that can serve both as a specification language and a language to reason about specifications
- the language needed to be easy to learn and build on things that programmers are familiar with:
  - familiar data structures
  - writing and executing a program
- Robust implementation – something that does not feel like a graduate student project 😊
Prolog as Proof Assistant

- Prolog fits the bill
  - designed as a programming language
  - rigorously based on first-order logic
  - uses a resolution based deduction engine (think automated modus ponens)
  - easy to learn
  - ISO standarized
  - lots of commercial and open source implementations available
  - I use SWI Prolog (www.swi-prolog.org)
Prolog as a Proof Assistant

- Downside:
  - no equational reasoning
    - writing a proof that $\sqrt{2}$ is irrational is difficult in Prolog
  - no type system
    - will not catch typos in term structures – difficult debugging
You just learned 90% of the Prolog language!
Prolog – Another Program

% recursive counting of elements
% in a list.

% base case:
% the count of an empty list is 0
count([],0).

% recursive step:
% the count of any list List is Count if
% List can be divided into a First element and the Rest of the list and
% T is the count of the Rest of the list and
% Count is T plus 1.
count(List,Count) :-
    List=[ First | Rest ],
    count(Rest,T),
    Count is T + 1.

% try it!
:- count([1,2,3],P),writeln(P).
Prolog as a Theorem Prover

- We have developed a library that makes Prolog deductions sound but incomplete
  - This is OK because we are using it as a proof assistant – only soundness is required.
  - interesting side node – with a little bit of work Prolog could be made quasi-complete
- Our library makes Prolog easy to use as a TP
Semantic Specifications 101

- We will define a simple calculator like language
- build a first-order logic model
- and then reason about the model
Semantic Specifications 101

% syntax definition -- Lisp like prefix notation for expressions
%
%  E ::= X
%  |  L
%  |  mult(E,E)
%  |  plus(E,E)
%  |  minus(E,E)
%
%  S ::= assign(X,E)
%  |  print(E)
%  |  S @ S
%
%  L ::= <any integer digit>
%  X ::= <any variable name>

Example: assign(x,plus(10,1)) @ print(x)
A simple ‘abstract interpreter’ model

The main operator in a semantic specification is the ‘maps to’ operator -->>

This operator maps a piece of syntax into its semantic domain (under the possible context of a state – the ‘::’ part)

```
% semantic definition of integer expressions

L --> L :-
   is_int(L),!.

B:: X --> V :-
   is_var(X),
   lookup(X,B,V),!.

B:: mult(E1,E2) --> V :-
   B:: E1 --> V1,
   B:: E2 --> V2,
   V xis V1 * V2,!.

B:: plus(E1,E2) --> V :-
   B:: E1 --> V1,
   B:: E2 --> V2,
   V xis V1 + V2,!.

B:: minus(E1,E2) --> V :-
   B:: E1 --> V1,
   B:: E2 --> V2,
   V xis V1 - V2,!.
```
Given a state the semantic value of a statement is another state!

% semantic definition of statements

\[
\begin{align*}
\text{B}:: \text{assign}(X,E) & \rightarrow \rightarrow \begin{cases} (X,V) \mid B \end{cases} :- \text{is\_var}(X), \\
& \text{B}:: E \rightarrow \rightarrow V,!. \\
\text{B}:: \text{print}(E) & \rightarrow \rightarrow B :- \\
& \text{B}:: E \rightarrow \rightarrow V, \\
& \text{write('Output value: ')} , \\
& \text{writeln}(V),!. \\
\text{B}:: S1 @ S2 & \rightarrow \rightarrow B2 :- \\
& \text{B}:: S1 \rightarrow \rightarrow B1, \\
& B1:: S2 \rightarrow \rightarrow B2,!. \\
\end{align*}
\]

% for convenience make '@' infix and left associative
:- op(725,yfx,@).

That’s it!
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('calc-sem.pl').
%  xis.pl compiled 0.00 sec, 33 clauses
%  preamble.pl compiled 0.01 sec, 45 clauses
%  calc-sem.pl compiled 0.01 sec, 58 clauses
true.

?- s:: assign(x,plus(10,1)) @ print(x) -->> S.
Output value: 11
S = [ (x, 11)|s].

?-
Proof – Semantic Equivalence

:- >>> 'Show that mult(2,3) is semantically equiv to add(3,3),'.
:- >>> 'it suffices to show that'.
:- >>> '(forall s)(exists V)'.
:- >>> '[s:: mult(2,3)-->>V ^ s:: plus(3,3)-->>V]'.

% proof
:- show s:: mult(2,3)-->>V , s:: plus(3,3)-->>V.
Proof – All Programs Terminate

- This is obvious because our language does not have function calls or loops
- but it still nice to actually prove it!
- The proof will show that the execution of any and every program will result in a value.
- Because syntactic domains can be viewed as inductively defined sets we can use induction to prove this.
Proof – All Programs Terminate

- Recall our syntax:

```plaintext
% syntax definition -- Lisp like prefix notation for expressions
%
%  E ::= X
%      | L
%      | mult(E,E)
%      | plus(E,E)
%      | minus(E,E)
%
%  S ::= assign(X,E)
%      | print(E)
%      | S @ S
%
%  L ::= <any integer digit>
%  X ::= <any variable name>
```
Proof – All Programs
Terminate

:- >>> 'induction on expressions'.

:- >>> 'Base cases:'.

:- >>> 'Variables'.
:- >>> 'Assume that states are finite'.
:- assume lookup(x,s,vx).
:- show s:: x ---> vx.

:- >>> 'Constants'.
:- assume is_int(n).
:- show s:: n ---> n.

:- >>> 'Inductive cases'.

:- >>> 'Operators'.
:- >>> 'mult'.
:- assume s:: a ---> va.
:- assume s:: b ---> vb.
:- show s:: mult(a,b) ---> va*vb.

:- >>> 'the remaining operators'.
:- >>> 'can be proved similarly'.

:- >>> 'induction on programming constructs'.

:- >>> 'Base cases:'.

:- >>> 'assignments'.
:- assume s:: e ---> ve.
:- show s:: assign(x,e) ---> [(x,ve)|s].

:- >>> 'print'.
:- assume s:: e ---> ve.
:- show s:: print(e) ---> s.

:- >>> 'Inductive step:'.

:- >>> 'composition'.
:- assume _A:: s1 ---> v1.
:- assume _B:: s2 ---> v2.
:- show s:: s1 @ s2 ---> v2.
Conclusions

- Fully automatic TP seems to be doomed because of the semi-decidability and NP trap
- Collaborative ATP or Proof Assistants build on the strengths of the structured approach humans take to theorem proving
- Collaborative ATP or Proof Assistants are versatile; going beyond mathematical theorem proving -- we have hardware verification, programming language semantics, etc.
- We are interested in Prolog as a theorem prover because of its simplicity, robustness, and availability – easy to learn – interesting as a first step into the theorem proving arena
Shameless Plug

- If you are interested in a mathematical approach to programming languages and theorem proving...
- ...I teach a course in programming language semantics which applies some of the things we saw here and more – CSC501
Thank You!

- Slides and Prolog code available at my homepage:
  - http://homepage.cs.uri.edu/faculty/hamel/