Automated Theorem Proving - summary of lecture 1

1 Introduction

Automated Theorem Proving (ATP) deals with the development of computer programs that show that some statement is a logical consequence of a set of statements. ATP systems are used in a wide variety of domains. For example, a mathematician might prove the conjecture that groups of order two are commutative, from the axioms of group theory; a management consultant might formulate axioms that describe how organizations grow and interact, and from those axioms prove that organizational death rates decrease with age; a hardware developer might validate the design of a circuit by proving a conjecture that describes a circuit’s performance, given axioms that describe the circuit itself. All of these are tasks that can be performed by an ATP system, given an appropriate formulation of the problem as axioms, hypotheses, and a conjecture.

The input of an ATP is a theorem. Examples of theorems that can be given to an ATP are:

1. Pythagoras theorem: Given a right triangle with sides A B and C, where C is the hypotenuse, then \( C^2 = A^2 + B^2 \).

2. Fundamental theorem of arithmetic: Any whole number bigger than 1 can be represented in exactly one way as a product of primes.

The theorem must be stated in formal logic. Many different kinds of logics are used: propositional, first-order, and also non-classical logics and higher-order logics. The logical language allows for a precise formal statement of the necessary information, which can then be manipulated by an ATP system. This formality is the underlying strength of ATP: there is no ambiguity in the statement of the problem, the problem is self-contained and no hidden assumptions are allowed (which is not always the case in theorems stated in math using a natural language). Users have to describe the problem at hand precisely and accurately, and this process in itself can lead to a clearer understanding of the problem domain. This in turn allows the user to formulate their problem appropriately for submission to an ATP system.

Some ATPs require human assistance, usually the user provides some guidance...
or hints, or tries to prove some intermediate results. The output of an ATP can be as simple as a yes/no answer, or it may include detailed proofs and/or counterexamples. The provided proofs are formal, unlike many proofs of mathematicians, which are formulated in a natural language, are usually validated by peer review and are meant to convey an intuition of how the proof works – for this purpose the formal details are too cumbersome. One of the most important requirements from an ATP is soundness - i.e., if the answer is positive, then the statement is true. If an ATP is also complete, then a negative answer means that the statement is indeed not true. ATPs first strive for soundness, and then for completeness if possible. Some ATPs are incomplete: no answer doesn’t provide any information. There are also many subtle variants: an ATP can be refutation complete - complete for providing an counterexample, or have a complete semi-algorithm - that is, complete but not always terminating.

2 Some history

In 1929 M. Presburger had shown that the first-order theory of addition in the arithmetic of integers is decidable, that is he had provided an algorithm which would be able to determine for a given sentence of that language, whether or not it is true. In 1954, M. Davis programmed this algorithm for the vacuum tube computer in Princeton. The algorithm performed very slowly. In the 1960s the field of ATP started to develop: the SAT problem and reductions to SAT were studied in the early 60s, Robinson introduced the method of resolution in 1965. In the 1970s some of the original excitement settles down, with the realization that the “really interesting” theorems are hard to prove automatically. Over the next three decades, the following large systems are developed:

- **Otter**
  Otter is designed to prove theorems stated in first-order logic with equality. Otter’s inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs.

- **Boyer-Moore**
  It was originally a fully-automatic theorem prover for a logic based on a home-grown dialect of Pure Lisp. The key ideas in the theorem prover were: the use of Lisp as a working logic; the reliance on a principle of definition for total recursive functions; the extensive use of rewriting and
“symbolic evaluation”; an induction heuristic based the failure of symbolic evaluation.

- **Coq**
  Coq is based on a logical framework called "Calculus of Inductive Constructions" extended by a modular development system for theories. In particular, Coq allows: the definition of functions or predicates; to state mathematical theorems and software specifications; to develop interactively formal proofs of these theorems; to check these proofs by a small certification "kernel".


In 1979 DeMillo, Lipton and Perlis argue that software verification is doomed:

It is argued that formal verifications for programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore, the absence of contitunity, the inevitability of change, and the complexity of specifications of significantly many real programs make the formal verification progress difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

However, currently, with the growth of the internet and the improvement of the state of the art, optimism towards ATP is growing again: technological readiness and an increased cost of bugs leads to a renewed interest in software verification, and the use of analysis techniques and/or theorem provers to verify software.

Some recent uses of theorem provers are:

1. **ECS/Java** - The Extended Static Checker for Java is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. Users can control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments. Extended Static Checking is a static analysis technique that relies on automated theorem proving. Its goals are to prevent common errors in programs, to make it easier to write reliable programs, and to increase programmer productivity. Verification of general properties of programs may require a lot of interaction with the programmer: he may need to enter assertions (like loop invariants). The tool does not require the whole program as an input, as it works in a modular way.
   ASC/Java tries to achieve a balance between the properties it covers and
the effort required by the programmer. In this sense it is different from the
more ambitious program verification tools based on Hoare logic and auto-
mated theorem provers. A detailed introduction of the tool can be found at

2. **SLAM** - The goal of the SLAM project of Microsoft (the tool was not
written by Microsoft) is to model check software written in common pro-
gramming languages directly. SLAM has been successfully used to find
and fix errors in Windows device drivers written in the C language. SLAM
operates by automatically constructing and refining model-checkable ab-
stractions (called Boolean Programs) of a C program. The tool is not
sound and not complete. (http://research.microsoft.com/slam/)

3. **Verisoft** - this is a long-term research project funded by the German
Federal Ministry of Education and Research. Project administrating or-
ganization is the german centre for Air- and Space Technology (DLR).
The main goal of the project is the pervasive formal verification of com-
puter systems. The correct functionality of systems, as they are applied,
for example, in automotive engineering, in security technology and in
the sector of medical technology, are to be mathematically proved. The
proofs are computer aided in order to prevent human error conducted
by the scientists involved. The knowledge and progress obtained are
expected to assist german enterprise in achieving a stable, internation-
ally competitive position in the professional spheres mentioned above.
(http://www.verisoft.de/index_en.html)

### 3 Consequence relations

A *consequence relation* is one of the most fundamental concepts in logic.

**Definition 1** A Taskian consequence relation ($\vdash_{\text{ter}}$) is a binary relation be-
tween sets of formulas and formulas of a language $L$, which satisfies the following
properties:

- $\{A\} \vdash A$
- If $\Gamma \vdash A$, then $\Gamma \cup \Delta \vdash A$.
- If $\Gamma \vdash A$ and $\Delta \cup \{A\} \vdash B$, then $\Gamma \cup \Delta \vdash B$.

**Definition 2** A Scott consequence relation ($\vdash_{\text{scr}}$) is a binary relation between
sets of formulas of a language $L$, which satisfies the following properties:

- $(\Gamma \cup \{A\}) \vdash (\Delta \cup A)$
- If $\Gamma \vdash \Delta$, then $\Gamma \cup \Gamma' \vdash \Delta \cup \Delta'$.
• If $\Gamma_1 \vdash \Delta_1 \cup \{A\}$ and $\Gamma_2 \vdash \Delta_2$, then $\Gamma_1 \cup \Gamma_2 \vdash \Delta_1 \cup \Delta_2$.

The first and second properties of an tcr/scr are usually referred to as reflexivity and monotonicity respectively. The second property is called transitivity or cut, and it captures the notion of introducing intermediate lemmas in a proof.

There are two important consequence relations used in first-order logic. In what follows, $L$ is a first-order language.

**Definition 3** 1. For a structure $M$ for $L$ and a valuation $v$ in $M$, $(M, v)$ is a t-model of a formula $A$ (a set of formulas $\Gamma$) if $M, v \models A$ for every $A \in \Gamma$.

2. A structure $M$ is a v-model of a formula $A$ (a set of formulas $\Gamma$) if for every valuation $v$ in $M$: $M, v \models A$ for every $A \in \Gamma$.

3. $T \vdash^t A$ if every t-model of $T$ is a t-model of $A$.

4. $T \vdash^v A$ if every v-model of $T$ is a v-model of $A$.

It is important to note that $\vdash^t$ and $\vdash^v$ are not identical. For instance, the generalization rule is valid for $\vdash^v$ but not for $\vdash^t$, that is $p(x) \vdash^v \forall x p(x)$, but $p(x)\vdash^t \forall x p(x)$. On the other hand, the classical deduction theorem holds for $\vdash^t$ but not for $\vdash^v$, that is $\Gamma \cup \{A\} \vdash^t B$ implies $\Gamma \vdash^t A \rightarrow B$, but $\Gamma \cup \{A\} \vdash^v B$ does not imply $\Gamma \vdash^v A \rightarrow B$. These relations are, however, identical form the point of view of theoremhood: $\vdash^t A$ iff $\vdash^v A$. Moreover, if the formulas of $\Gamma$ are all closed (i.e, no variables occur free in them), then $\vdash^t A$ iff $\vdash^v A$.

Different proof systems define different consequence relations. For instance, the Natural Deduction system for first-order classical logic uses the $\vdash^t$ relation, while the resolution calculus uses the $\vdash^v$ relation.

## 4 Natural Deduction

We start with a proof system which is designed to capture the way that mathematicians construct their proofs.

The system $\text{NDFOL}$:

\[
\begin{align*}
(ax) \{A\} \cup \Gamma & \Rightarrow A \\
\Gamma_1 \Rightarrow A & \quad \Gamma_2 \Rightarrow B \quad \Gamma_1 \cup \Gamma_2 \Rightarrow A \land B \\
\Gamma_1 \cup \Gamma_2 \Rightarrow A \land B & \quad \Gamma \Rightarrow A \land B \quad \Gamma \Rightarrow B \\
\Gamma \Rightarrow A \land B & \quad (\land E_1) \quad \Gamma \Rightarrow A \land B \quad (\land E_2) \\
\Gamma \Rightarrow A \land B & \quad \Gamma \Rightarrow A \land B \\
\Gamma \Rightarrow A \land B & \quad (\rightarrow I) \quad \Gamma_1 \Rightarrow A \quad \Gamma_2 \Rightarrow B \quad \Gamma_1 \cup \Gamma_2 \Rightarrow A \rightarrow B \\
\Gamma_1 \Rightarrow A \land B & \quad \Gamma_2 \Rightarrow A \rightarrow B \\
\Gamma_1 \cup \Gamma_2 \Rightarrow A \land B & \quad (\rightarrow E) \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \Rightarrow A & \quad \text{(VI)} \\
\Gamma \Rightarrow A \lor B & \quad \Gamma \Rightarrow B \
\Gamma \Rightarrow A \lor B & \quad \text{(VI)} \\
\Gamma_1 \Rightarrow A \lor B & \quad \{A\} \cup \Gamma_2 \Rightarrow C \\
\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \Rightarrow C & \quad \text{(\lor E)} \\
\{A\} \cup \Gamma_1 \Rightarrow B & \quad \{A\} \cup \Gamma_2 \Rightarrow \neg B \\
\Gamma_1 \cup \Gamma_2 \Rightarrow \neg A & \quad \text{(\neg I)} \\
\Gamma \Rightarrow \neg\neg A & \quad \text{(\neg\neg E)} \\
\Gamma \Rightarrow \neg\neg A & \quad \Gamma \Rightarrow A \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \Rightarrow A(y/x) & \quad (**) (\forall I) \\
\Gamma \Rightarrow \forall x A & \quad \Gamma \Rightarrow \forall x A \quad \text{\Gamma \Rightarrow A(t/x)} \\
\Gamma \Rightarrow A(t/x) & \quad (**) (\forall E) \\
\Gamma \Rightarrow \exists x A & \quad \text{\Gamma \Rightarrow \exists x A \quad \Gamma, A(y/x) \Rightarrow C} \\
\Gamma \Rightarrow C & \quad (**) (\exists E)
\end{align*}
\]

(*) - \( t \) is free for \( x \) in \( A \)

(**) - \( y \) is free for \( x \) in \( A \) and does not occur free in \( \Gamma \cup \{\exists x A, C\} \)