Theorem Proving: Introduction

Applications of Theorem Proving

- Logic is used for reasoning about programs. complete correctness proofs, or proofs of absence of crashes.
- Propositional logic is used for reasoning about boolean circuits.
- It has been used used for the verification of mathematical results like the 4-color theorem, a proof of the Kepler conjecture is being formalized. The main theorem of algebra has been formalized. Large parts of high-school calculus have been formalized.
- It is used for databases. (declarative databases)
- Natural language processing. (Anaphora resolution, question answering)
- Search engines, ontologies, XML.

Current Research

- Theory and implementation of automated proof search. (in first-order logic, in logic with theories)
- Building up experience in the verification of programs and mathematics.
- Logic support during the development of programs.
- Finding good logics for ontologies.
- Integration of Common Basic Types into logic (SMT).

Topics of the Lecture

- Propositional Logic. Sequent calculus for propositional Logic.
- Semantics for propositional logic, completeness and soundness of sequent calculus.

  Theorem Proving with propositional logic. Circuit Verification. Solving Sudokus. Alphametic Puzzles
- Higher-Order Logic. Formalization of Mathematics in HOL. Correctness proof of Unification in HOL system.
- Curry-Howard isomorphism, COQ system.

- Theorem Proving in First-Order Logic, Skolemization, Normal Forms.
- Implementation of Resolution. Data structures.
- CASC competition.
- Geometric Logic, Partial Functions, Kleene logic.

When selecting a proof checker, or designing one, the following questions are important:

Classical or Intuitionistic?

Intuitionistic proofs have more meaning than classical proofs. In general, when an intuitionistic proof promises that something exists, it is possible to compute the object that is promised. This means that the proof contains an algorithm. Intuitionistic proofs are harder to find. Some theorems may have a classical proof, but not an intuitionistic proof.

I do not believe in intuitionistic logic as a tool in software construction, because it breaks fundamental rules of software development: Specification should be separated from implementation. In addition to that, the underlying algorithmic calculus is too rigid to be useful (only primitive programs with functional recursion are possible.) Still, it is very elegant and fundamental, so it should be taught. Knowing about intuitionistic logic improves the quality of your proofs.

Level of Automation

How much can the system do automatic? If you are designing a calculus, how suitable is this calculus for automatic treatment?

There exist pretty strong first-order theorem provers, but these systems are very sensitive to small perturbations: They cannot handle types, not even simple types. They cannot handle any form of higher-order syntax, or primitive data types, even when they are allowed to ignore them.

Do you want to verify programs or proofs? If you want to verify programs, you will want automatic treatment of primitive types. (integers, reals, etc.)

Some calculi allow definition of logical operators (or, and, exists, equality). This is theoretically very elegant, but once the operators are removed, they are not recognizable anymore by an automatic system.

Proofs and Trust

How much code in the system needs to be trusted? Ideally, the system should generate proofs in a simple calculus, that can be checked independently by a simple program.

In practice, one needs to trust the OS, the memory, the processor, and the compiler. If the proofs can be checked by different programs on different computers, this is not a real issue.

If no proofs can be generated, the system may use the HOL(light) approach. In the HOL approach, constructed proofs are checked internally by defining a class **theorem** with constructors that correspond to logical proof rules. This implies that every object that with type **theorem** object is indeed a theorem.

This approach is still quite good, but proofs cannot be checked independently. The HOL approach was developed in early 1970-ies, because memory was expensive in those days. (50 groszy/bit.)

Proofs and Trust (2)

Proofs and trust are an issue with theorem provers. First-order theorem provers don't output proofs, and are changing all the time. Together with the interface problems discussed before, it can be concluded that usage of first-order theorem provers in verification is problematic.

Treatment of Definitions

If you are designing a calculus, you need to answer the following question: How to handle definitions?

Do you want a dedicated mechanism, or to use equality/equivalence? Defining new predicates is not a real problem.

The real problem is with the definition of functions: Introduce from $\forall x \exists y\ P(x, y)$ implies that function f chooses. This is not conservative, while definitions are supposed to be conservative (i.e. they can be easily eliminated.) Axiom of choice (or ϵ axiom) is convenient but controversial.

Better is using the condition $\forall x \exists y\ P(x, y) \wedge \forall y\ P(x, y') \rightarrow y \approx y'$. I prefer this approach, but HOL uses the ϵ rule.

Type System

Which type system to use? If you don't define a type system in the logic, you have to make types explicit, using $\rightarrow$ and $\wedge$. This gets very quickly very unpleasant. In addition, it doesn't help you with checking type correctness. (Untyped, first-order logic is like assembler.)

Simple types are better, but still sometimes too weak. Polymorphic types are better. Overlapping types? (Nice, but hard to check.)

Automatic introduction of inductive types is useful, but difficult to implement. HOL has it, and it works very good. COQ also has it, but I don't like it.

Implementation of Substitution

If you are going to implement something, then you have to decide how to implement local variables and substitution. De Bruijn-indices, or named variables? Both have their problems. Named variables carry meaning, but their treatment is computationally costly.

De Bruijn indices are technically simpler, but formulas become tied to their defining contexts. This makes implementation tricky. If you wouldn't do that, abstraction becomes costly and computational advantage is lost.

Still I think that de Bruijn indices with lazy substitution are the best way.

Three Main Approaches to Mathematical Proofs

We will study calculi in which mathematical proofs can be represented, and automatically verified. As far as I know, there are three reasonable approaches to the verification of mathematical proofs:

1. First-Order logic + Set Theory. The advantage of this approach is its techical simplicity. It is easy to define and to implement. Unfortunately, in bigger proofs, set theory behaves like assembler language. It is too low level (untyped).
2. Higher-Order Logic. Higher-Order Logic is reasonably simple, easy to implement, and powerful enough to be practically useful.
3. Intuitionistic Logic with Curry-Howard isomorphism.