Theorem Proving for Verification
1: Background & Propositional Logic

John Harrison

Intel Corporation

Marktoberdorf 2010

Wed 11th August 2010 (08:45–09:30)

Plan for the lectures

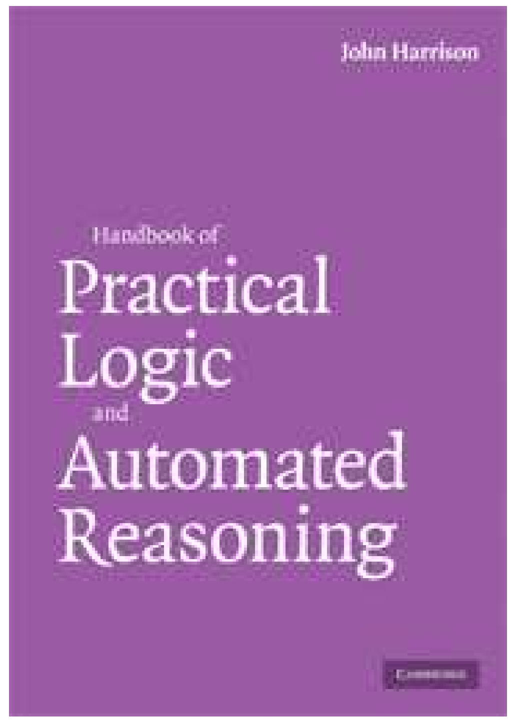
Some of the main techniques for automated theorem proving, as applied in verification.

1. Propositional logic (SAT)
2. First-order logic and arithmetical theories
3. Combination and certification of decision procedures (SMT)
4. - EITHER Cohen-Hörmander real quantifier elimination
   - OR Interactive theorem proving

For more details

An introductory survey of many central results in automated reasoning, together with actual OCaml model implementations

`http://www.cl.cam.ac.uk/~jrh13/atp/index.html`



We probably all know what propositional logic is.

English	Standard	Boolean	Other
false	$\bot$	0	F
true	$\top$	1	T
not p	$\neg p$	$\overline{p}$	$-p$, $\sim p$
p and q	$p \wedge q$	pq	$p \& q$, $p \cdot q$
p or q	$p \vee q$	$p + q$	$p \mid q$, $p\ or\ q$
p implies q	$p \Rightarrow q$	$p \leqslant q$	$p \rightarrow q$, $p \supset q$
p iff q	$p \Leftrightarrow q$	$p = q$	$p \equiv q$, $p \sim q$

In the context of circuits, it's often referred to as 'Boolean algebra', and many designers use the Boolean notation.

Is propositional logic boring?

Traditionally, propositional logic has been regarded as fairly boring.

- There are severe limitations to what can be said with propositional logic.
- Propositional logic is trivially decidable in theory.
- Propositional satisfiability (SAT) is the original NP-complete problem, so seems intractible in practice.

But . . .

No!

The last decade or so has seen a remarkable upsurge of interest in propositional logic.

Why the resurgence?

No!

The last decade or so has seen a remarkable upsurge of interest in propositional logic.

Why the resurgence?

- There *are* many interesting problems that can be expressed in propositional logic
- Efficient algorithms *can* often decide large, interesting problems of real practical relevance.

The many applications almost turn the ‘NP-complete’ objection on its head.

Logic and circuits

The correspondence between digital logic circuits and propositional logic has been known for a long time.

Digital design	Propositional Logic
circuit	formula
logic gate	propositional connective
input wire	atom
internal wire	subexpression
voltage level	truth value

Many problems in circuit design and verification can be reduced to propositional tautology or satisfiability checking ('SAT').

For example optimization correctness: $\phi \Leftrightarrow \phi'$ is a tautology.

Combinatorial problems

Many other apparently difficult combinatorial problems can be encoded as Boolean satisfiability, e.g. scheduling, planning, geometric embeddibility, even factorization.

$$
\begin{aligned}
\neg(\quad & (out_0 \Leftrightarrow x_0 \wedge y_0) \wedge \\
& (out_1 \Leftrightarrow (x_0 \wedge y_1 \Leftrightarrow \neg(x_1 \wedge y_0))) \wedge \\
& (v_2^2 \Leftrightarrow (x_0 \wedge y_1) \wedge x_1 \wedge y_0) \wedge \\
& (u_2^0 \Leftrightarrow ((x_1 \wedge y_1) \Leftrightarrow \neg v_2^2)) \wedge \\
& (u_2^1 \Leftrightarrow (x_1 \wedge y_1) \wedge v_2^2) \wedge \\
& (out_2 \Leftrightarrow u_2^0) \wedge (out_3 \Leftrightarrow u_2^1) \wedge \\
& \neg out_0 \wedge out_1 \wedge out_2 \wedge \neg out_3)
\end{aligned}
$$

Read off the factorization $6 = 2 \times 3$ from a refuting assignment.

Efficient methods

The naive truth table method is quite impractical for formulas with more than a dozen primitive propositions.

Practical use of propositional logic mostly relies on one of the following algorithms for deciding tautology or satisfiability:

- Binary decision diagrams (BDDs)
- The Davis-Putnam method (DP, DPLL)
- Stålmarck's method

We'll sketch the basic ideas behind Davis-Putnam.

DP and DPLL

Actually, the original Davis-Putnam procedure is not much used now.

What is usually called the Davis-Putnam method is actually a later refinement due to Davis, Loveland and Logemann (hence DPLL).

We formulate it as a test for *satisfiability*. It has three main components:

- Transformation to conjunctive normal form (CNF)
- Application of simplification rules
- Splitting

Normal forms

In ordinary algebra we can reach a 'sum of products' form of an expression by:

- Eliminating operations other than addition, multiplication and negation, e.g. $x - y \mapsto x + -y$.
- Pushing negations inwards, e.g. $-(-x) \mapsto x$ and $-(x + y) \mapsto -x + -y$.
- Distributing multiplication over addition, e.g. $x(y + z) \mapsto xy + xz$.

In logic we can do exactly the same, e.g. $p \Rightarrow q \mapsto \neg p \vee q$, $\neg(p \wedge q) \mapsto \neg p \vee \neg q$ and $p \wedge (q \vee r) \mapsto (p \wedge q) \vee (p \wedge r)$.

The first two steps give 'negation normal form' (NNF).

Following with the last (distribution) step gives 'disjunctive normal form' (DNF), analogous to a sum-of-products.

Conjunctive normal form

Conjunctive normal form (CNF) is the dual of DNF, where we reverse the roles of 'and' and 'or' in the distribution step to reach a 'product of sums':

$$
\begin{array}{rcl}
p \lor (q \land r) & \mapsto & (p \lor q) \land (p \lor r) \\
(p \land q) \lor r & \mapsto & (p \lor r) \land (q \lor r)
\end{array}
$$

Reaching such a CNF is the first step of the Davis-Putnam procedure.

Conjunctive normal form

Conjunctive normal form (CNF) is the dual of DNF, where we reverse the roles of ‘and’ and ‘or’ in the distribution step to reach a ‘product of sums’:

$$\begin{aligned} p \lor (q \land r) &\quad\mapsto\quad (p \lor q) \land (p \lor r) \\ (p \land q) \lor r &\quad\mapsto\quad (p \lor r) \land (q \lor r) \end{aligned}$$

Reaching such a CNF is the first step of the Davis-Putnam procedure.

Unfortunately the naive distribution algorithm can cause the size of the formula to grow exponentially — not a good start. Consider for example:

$$(p_1 \land p_2 \land \cdots \land p_n) \lor (q_1 \land p_2 \land \cdots \land q_n)$$

Definitional CNF

A cleverer approach is to introduce new variables for subformulas. Although this isn't logically equivalent, it does preserve satisfiability.

$$(p \lor (q \land \neg r)) \land s$$

introduce new variables for subformulas:

$$(p_1 \Leftrightarrow q \land \neg r) \land (p_2 \Leftrightarrow p \lor p_1) \land (p_3 \Leftrightarrow p_2 \land s) \land p_3$$

then transform to (3-)CNF in the usual way:

$$(\neg p_1 \lor q) \land (\neg p_1 \lor \neg r) \land (p_1 \lor \neg q \lor r) \land$$

$$(\neg p_2 \lor p \lor p_1) \land (p_2 \lor \neg p) \land (p_2 \lor \neg p_1) \land$$

$$(\neg p_3 \lor p_2) \land (\neg p_3 \lor s) \land (p_3 \lor \neg p_2 \lor \neg s) \land p_3$$

Clausal form

It's convenient to think of the CNF form as a set of sets:

- Each disjunction $p_1 \vee \cdots \vee p_n$ is thought of as the set $\{p_1, \ldots, p_n\}$, called a *clause*.
- The overall formula, a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$ is thought of as a set $\{C_1, \ldots, C_m\}$.

Since 'and' and 'or' are associative, commutative and idempotent, nothing of logical significance is lost in this interpretation.

Special cases: an empty clause means $\bot$ (and is hence unsatisfiable) and an empty set of clauses means $\top$ (and is hence satisfiable).

Simplification rules

At the core of the Davis-Putnam method are two transformations on the set of clauses:

I The 1-literal rule: if a unit clause p appears, remove $\neg p$ from other clauses and remove all clauses including p.

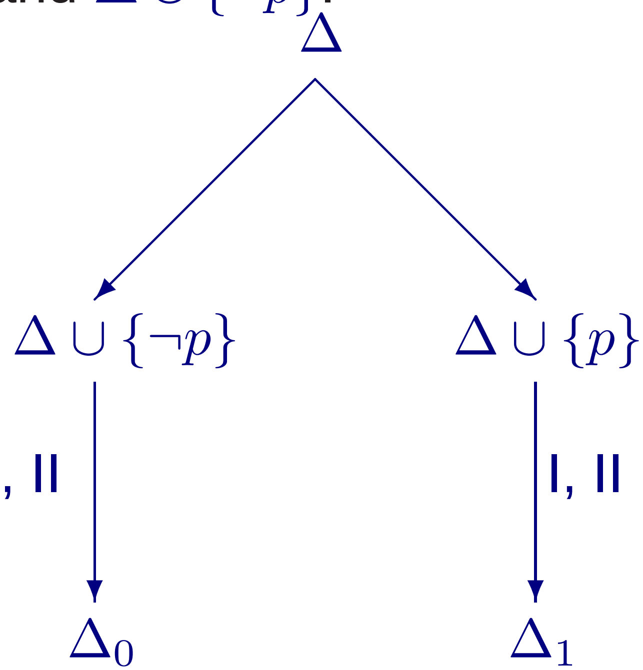
II The affirmative-negative rule: if p occurs *only* negated, or *only* unnegated, delete all clauses involving p.

These both preserve satisfiability of the set of clause sets.

Splitting

In general, the simplification rules will not lead to a conclusion. We need to perform case splits.

Given a clause set Δ, simply choose a variable p, and consider the two new sets $\Delta \cup \{p\}$ and $\Delta \cup \{\neg p\}$.



In general, these case-splits need to be nested.

DPLL completeness

Each time we perform a case split, the number of unassigned literals is reduced, so eventually we must terminate. Either

- For all branches in the tree of case splits, the empty clause is derived: the original formula is unsatisfiable.
- For some branch of the tree, we run out of clauses: the formula is satisfiable.

In the latter case, the decisions leading to that leaf give rise to a satisfying assignment.

Modern SAT solvers

Much of the improvement in SAT solver performance in recent years has been driven by several improvements to the basic DPLL algorithm:

- Non-chronological backjumping, learning conflict clauses
- Optimization of the basic ‘constraint propagation’ rules (“watched literals” etc.)
- Good heuristics for picking ‘split’ variables, and even restarting with different split sequence
- Highly efficient data structures

Some well-known SAT solvers are Chaff, MiniSat and PicoSAT.

Backjumping motivation

Suppose we have clauses

$$\neg p_1 \vee \neg p_{10} \vee p_{11}$$

$$\neg p_1 \vee \neg p_{10} \vee \neg p_{11}$$

If we split over variables in the order p_1,...,p_{10}, assuming first that they are true, we then get a conflict.

Yet none of the assignments to p_2,...,p_9 are relevant.

We can backjump to the decision on p_1 and assume $\neg p_{10}$ at once.

Or backtrack all the way and add $\neg p_1 \vee \neg p_{10}$ as a deduced 'conflict' clause.

Summary

- Propositional logic is no longer a neglected area of theorem proving
- A wide variety of practical problems can usefully be encoded in SAT
- There is intense interest in efficient algorithms for SAT
- Many of the most successful systems are still based on refinements of the ancient Davis-Putnam procedure

Theorem Proving for Verification
2: First-order logic and arithmetical theories

John Harrison

Intel Corporation

Marktoberdorf 2010

Thu 12th August 2010 (10:45 – 11:30)

Summary

- First order logic
- Naive Herbrand procedures
- Unification
- Decidable classes
- Decidable theories
- Quantifier elimination

First-order logic

Start with a set of *terms* built up from variables and constants using function application:

$$x + 2 \cdot y \;\; \equiv \;\; +(x, \cdot(2(), y))$$

Create atomic formulas by applying relation symbols to a set of terms

$$x > y \;\; \equiv \; > (x, y)$$

Create complex formulas using quantifiers

- $\forall x.\ P[x]$ — for all x, $P[x]$
- $\exists x.\ P[x]$ — there exists an x such that $P[x]$

Quantifier examples

The order of quantifier nesting is important. For example

$\forall x.\ \exists y.\ loves(x, y)$ — everyone loves someone
$\exists x.\ \forall y.\ loves(x, y)$ — somebody loves everyone
$\exists y.\ \forall x.\ loves(x, y)$ — someone is loved by everyone

This says that a function $\mathbb{R} \to \mathbb{R}$ is continuous:

$$\forall \epsilon.\ \epsilon > 0 \Rightarrow \forall x.\ \exists \delta.\ \delta > 0 \land \forall x'.\ |x' - x| < \delta \Rightarrow |f(x') - f(x)| < \varepsilon$$

while this one says it is *uniformly* continuous, an important distinction

$$\forall \epsilon.\ \epsilon > 0 \Rightarrow \exists \delta.\ \delta > 0 \land \forall x.\ \forall x'.\ |x' - x| < \delta \Rightarrow |f(x') - f(x)| < \varepsilon$$

Skolemization

Skolemization relies on this observation (related to the axiom of choice):

$$(\forall x.\ \exists y.\ P[x, y]) \Leftrightarrow \exists f.\ \forall x.\ P[x, f(x)]$$

For example, a function is surjective (onto) iff it has a right inverse:

$$(\forall x.\ \exists y.\ g(y) = x) \Leftrightarrow (\exists f.\ \forall x.\ g(f(x)) = x$$

Can't quantify over functions in first-order logic.

But we get an *equisatisfiable* formula if we just introduce a new function symbol.

$$\begin{aligned} &\forall x_1, \ldots, x_n.\ \exists y.\ P[x_1, \ldots, x_n, y] \\ &\rightarrow \forall x_1, \ldots, x_n.\ P[x_1, \ldots, x_n, f(x_1, \ldots, x_n)] \end{aligned}$$

Now we just need a satisfiability test for universal formulas.

First-order automation

The underlying domains can be arbitrary, so we can't do an exhaustive analysis, but must be slightly subtler.

We can reduce the problem to propositional logic using the so-called *Herbrand theorem*:

> Let $\forall x_1, \ldots, x_n.\ P[x_1, \ldots, x_n]$ be a first order formula with only the indicated universal quantifiers (i.e. the body $P[x_1, \ldots, x_n]$ is quantifier-free). Then the formula is satisfiable iff the infinite set of 'ground instances' $P[t_1^i, \ldots, t_n^i]$ that arise by replacing the variables by arbitrary variable-free terms made up from functions and constants in the original formula is *propositionally* satisfiable.

Still only gives a *semidecision* procedure, a kind of proof search.

Example

Suppose we want to prove the 'drinker's principle'

$$\exists x.\, \forall y.\, D(x) \Rightarrow D(y)$$

Negate the formula, and prove negation unsatisfiable:

$$\neg(\exists x.\, \forall y.\, D(x) \Rightarrow D(y))$$

Convert to prenex normal form: $\forall x.\, \exists y.\, D(x) \wedge \neg D(y)$

Skolemize: $\forall x.\, D(x) \wedge \neg D(f(x))$

Enumerate set of ground instances, first $D(c) \wedge \neg D(f(c))$ is not unsatisfiable, but the next is:

$$(D(c) \wedge \neg D(f(c))) \wedge (D(f(c)) \wedge \neg D(f(f(c)))$$

Unification

The first automated theorem provers actually used that approach.

It was to test the propositional formulas resulting from the set of ground-instances that the Davis-Putnam method was developed.

However, more efficient than enumerating ground instances is to use *unification* to choose instantiations intelligently.

Many theorem-proving algorithms based on unification exist:

- Tableaux
- Resolution
- Model elimination
- Connection method
- ...

Decidable problems

Although first order validity is undecidable, there are special cases where it is decidable, e.g.

- AE formulas: no function symbols, universal quantifiers before existentials in prenex form (so finite Herbrand base).
- Monadic formulas: no function symbols, only unary predicates

These are not particularly useful in practice, though they can be used to automate syllogistic reasoning.

If all M are P, and all S are M, then all S are P

can be expressed as the monadic formula:

$$(\forall x.\ M(x) \Rightarrow P(x)) \wedge (\forall x.\ S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\ S(x) \Rightarrow P(x))$$

The theory of equality

A simple but useful decidable theory is the universal theory of equality with function symbols, e.g.

$$\forall x.\ f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \Rightarrow f(x) = x$$

after negating and Skolemizing we need to test a ground formula for satisfiability:

$$f(f(f(c))) = c \wedge f(f(f(f(f(c))))) = c \wedge \neg(f(c) = c)$$

Two well-known algorithms:

- Put the formula in DNF and test each disjunct using one of the classic ‘congruence closure’ algorithms.
- Reduce to SAT by introducing a propositional variable for each equation between subterms and adding constraints.

Decidable theories

More useful in practical applications are cases not of *pure* validity, but validity in special (classes of) models, or consequence from useful axioms, e.g.

- Does a formula hold over all rings (Boolean rings, non-nilpotent rings, integral domains, fields, algebraically closed fields, . . .)
- Does a formula hold in the natural numbers or the integers?
- Does a formula hold over the real numbers?
- Does a formula hold in all real-closed fields?
- . . .

Because arithmetic comes up in practice all the time, there's particular interest in theories of arithmetic.

Theories

These can all be subsumed under the notion of a *theory*, a set of formulas T closed under logical validity. A theory T is:

- *Consistent* if we never have $p \in T$ and $(\neg p) \in T$.
- *Complete* if for closed p we have $p \in T$ or $(\neg p) \in T$.
- *Decidable* if there's an algorithm to tell us whether a given closed p is in T

Note that a complete theory generated by an r.e. axiom set is also decidable.

Quantifier elimination

Often, a quantified formula is T-equivalent to a quantifier-free one:

- $\mathbb{C} \models (\exists x.\ x^2 + 1 = 0) \Leftrightarrow \top$
- $\mathbb{R} \models (\exists x. ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geqslant 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$
- $\mathbb{Q} \models (\forall x.\ x < a \Rightarrow x < b) \Leftrightarrow a \leqslant b$
- $\mathbb{Z} \models (\exists k\ x\ y.\ ax = (5k + 2)y + 1) \Leftrightarrow \neg(a = 0)$

We say a theory T admits *quantifier elimination* if *every* formula has this property.

Assuming we can decide variable-free formulas, quantifier elimination implies completeness.

And then an *algorithm* for quantifier elimination gives a decision method.

Important arithmetical examples

- Presburger arithmetic: arithmetic equations and inequalities with addition but *not multiplication*, interpreted over $\mathbb{Z}$ or $\mathbb{N}$.
- Tarski arithmetic: arithmetic equations and inequalities with addition and multiplication, interpreted over $\mathbb{R}$ (or any real-closed field)
- General algebra: arithmetic equations with addition and multiplication interpreted over $\mathbb{C}$ (or other algebraically closed field).

However, arithmetic with multiplication over $\mathbb{Z}$ is not even semidecidable, by Gödel's theorem.

Nor is arithmetic over $\mathbb{Q}$ (Julia Robinson), nor just solvability of equations over $\mathbb{Z}$ (Matiyasevich). Equations over $\mathbb{Q}$ unknown.

Summary

- Can't solve first-order logic by naive method, but Herbrand's theorem gives a proof search procedure
- Unification is normally a big improvement on straightforward search through the Herbrand base
- A few fragments of first-order logic are decidable, but few are very useful.
- We are often more interested in arithmetic theories than pure logic
- Quantifier elimination usually gives a nice decision method and more

Theorem Proving for Verification
3: Combining and certifying decision procedures

John Harrison

Intel Corporation

Marktoberdorf 2010

Fri 13th August 2010 (09:35 – 10:20)

Summary

- Need to combine multiple decision procedures
- Basics of Nelson-Oppen method
- Proof-producing decision procedures
- Separate certification

Need for combinations

In applications we often need to combine decision methods from different domains.

$$x - 1 < n \land \neg(x < n) \Rightarrow a[x] = a[n]$$

An arithmetic decision procedure could easily prove

$$x - 1 < n \land \neg(x < n) \Rightarrow x = n$$

but could not make the additional final step, even though it looks trivial.

Most combinations are undecidable

Adding almost any additions, especially uninterpreted, to the usual decidable arithmetic theories destroys decidability.

Some exceptions like BAPA ('Boolean algebra + Presburger arithmetic').

This formula over the reals constrains P to define the integers:

$$(\forall n.\, P(n+1) \Leftrightarrow P(n)) \land (\forall n.\, 0 \leqslant n \land n < 1 \Rightarrow (P(n) \Leftrightarrow n = 0))$$

and this one in Presburger arithmetic defines squaring:

$$(\forall n.\, f(-n) = f(n)) \land (f(0) = 0) \land$$
$$(\forall n.\, 0 \leqslant n \Rightarrow f(n+1) = f(n) + n + n + 1)$$

and so we can define multiplication.

Quantifier-free theories

However, if we stick to so-called ‘quantifier-free’ theories, i.e. deciding universal formulas, things are better.

Two well-known methods for combining such decision procedures:

- Nelson-Oppen
- Shostak

Nelson-Oppen is more general and conceptually simpler.

Shostak seems more efficient where it does work, and only recently has it really been understood.

Nelson-Oppen basics

Key idea is to combine theories $T_1, \ldots, T_n$ with *disjoint signatures*. For instance

- T_1: numerical constants, arithmetic operations
- T_2: list operations like cons, head and tail.
- T_3: other uninterpreted function symbols.

The only common function or relation symbol is '$=$'.

This means that we only need to share formulas built from equations among the component decision procedure, thanks to the *Craig interpolation theorem*.

The interpolation theorem

Several slightly different forms; we'll use this one (by compactness, generalizes to theories):

> If $\models \phi_1 \land \phi_2 \Rightarrow \bot$ then there is an 'interpolant' ψ, whose only free variables and function and predicate symbols are those occurring in *both* ϕ_1 and ϕ_2, such that $\models \phi_1 \Rightarrow \psi$ and $\models \phi_2 \Rightarrow \neg\psi$.

This is used to assure us that the Nelson-Oppen method is complete, though we don't need to produce general interpolants in the method.

In fact, interpolants can be found quite easily from proofs, including Herbrand-type proofs produced by resolution etc.

Nelson-Oppen I

Proof by example: refute the following formula in a mixture of Presburger arithmetic and uninterpreted functions:

$$f(v-1)-1 = v+1 \wedge f(u)+1 = u-1 \wedge u+1 = v$$

First step is to *homogenize*, i.e. get rid of atomic formulas involving a mix of signatures:

$$u+1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_2 = f(v_3) \wedge v_1 = f(u) \wedge v_3 = v - 1$$

so now we can split the conjuncts according to signature:

$$(u+1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_3 = v - 1) \wedge$$
$$(v_2 = f(v_3) \wedge v_1 = f(u))$$

Nelson-Oppen II

If the entire formula is contradictory, then there's an interpolant ψ such that in Presburger arithmetic:

$$\mathbb{Z} \models u + 1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_3 = v - 1 \Rightarrow \psi$$

and in pure logic:

$$\models v_2 = f(v_3) \wedge v_1 = f(u) \wedge \psi \Rightarrow \bot$$

We can assume it only involves variables and equality, by the interpolant property and disjointness of signatures.

Subject to a technical condition about finite models, the pure equality theory admits quantifier elimination.

So we can assume ψ is a propositional combination of equations between variables.

Nelson-Oppen III

In our running example, $u = v_3 \wedge \neg(v_1 = v_2)$ is one suitable interpolant, so

$\mathbb{Z} \models u + 1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_3 = v - 1 \Rightarrow u = v_3 \wedge \neg(v_1 = v_2)$

in Presburger arithmetic, and in pure logic:

$\models v_2 = f(v_3) \wedge v_1 = f(u) \Rightarrow u = v_3 \wedge \neg(v_1 = v_2) \Rightarrow \bot$

The component decision procedures can deal with those, and the result is proved.

Nelson-Oppen IV

Could enumerate all significantly different potential interpolants.

Better: case-split the original problem over all possible equivalence relations between the variables (5 in our example).

$$T_1, \ldots, T_n \models \phi_1 \wedge \cdots \wedge \phi_n \wedge ar(P) \Rightarrow \bot$$

So by interpolation there's a C with

$$T_1 \models \phi_1 \wedge ar(P) \Rightarrow C$$

$$T_2, \ldots, T_n \models \phi_2 \wedge \cdots \wedge \phi_n \wedge ar(P) \Rightarrow \neg C$$

Since $ar(P) \Rightarrow C$ or $ar(P) \Rightarrow \neg C$, we must have one theory with $T_i \models \phi_i \wedge ar(P) \Rightarrow \bot$.

Nelson-Oppen V

Still, there are quite a lot of possible equivalence relations ($\text{bell}(5) = 52$), leading to large case-splits.

An alternative formulation is to repeatedly let each theory deduce new disjunctions of equations, and case-split over them.

$$T_i \models \phi_i \Rightarrow x_1 = y_1 \vee \cdots \vee x_n = y_n$$

This allows two important optimizations:

- If theories are *convex*, need only consider pure equations.
- Component procedures can actually produce equational consequences rather than waiting passively for formulas to test.

Most SMT solvers use a SAT solver as a core and use the component decision procedures to produce new conflict clauses.

Certification of decision procedures

We might want a decision procedure to produce a ‘proof’ or ‘certificate’

- Doubts over the correctness of the core decision method
- Desire to use the proof in other contexts

This arises in at least two real cases:

- Fully expansive (e.g. ‘LCF-style’) theorem proving.
- Proof-carrying code

Certifiable and non-certifiable

The most desirable situation is that a decision procedure should produce a short certificate that can be checked easily.

Factorization and primality is a good example:

- Certificate that a number is not prime: the factors! (Others are also possible.)
- Certificate that a number is prime: Pratt, Pocklington, Pomerance, ...

This means that primality checking is in NP $\cap$ co-NP (we now know it's in P).

Certifying universal formulas over $\mathbb{C}$

Use the (weak) *Hilbert Nullstellensatz*:

The polynomial equations $p_1(x_1,\ldots,x_n) = 0$, ..., $p_k(x_1,\ldots,x_n) = 0$ in an algebraically closed field have *no* common solution iff there are polynomials $q_1(x_1,\ldots,x_n)$, ..., $q_k(x_1,\ldots,x_n)$ such that the following polynomial identity holds:

$$q_1(x_1,\ldots,x_n)\cdot p_1(x_1,\ldots,x_n)+\cdots+q_k(x_1,\ldots,x_n)\cdot p_k(x_1,\ldots,x_n) = 1$$

All we need to certify the result is the cofactors $q_i(x_1,\ldots,x_n)$, which we can find by an instrumented Gröbner basis algorithm.

The checking process involves just algebraic normalization (maybe still not totally trivial. . .)

Certifying universal formulas over $\mathbb{R}$

There is a similar but more complicated Nullstellensatz (and Positivstellensatz) over $\mathbb{R}$.

The general form is similar, but it's more complicated because of all the different orderings.

It inherently involves sums of squares (SOS), and the certificates can be found efficiently using semidefinite programming (Parillo . . .)

Example: easy to check

$$\forall a\ b\ c\ x.\ ax^2 + bx + c = 0 \Rightarrow b^2 - 4ac \geqslant 0$$

via the following SOS certificate:

$$b^2 - 4ac = (2ax + b)^2 - 4a(ax^2 + bx + c)$$

Less favourable cases

Unfortunately not all decision procedures seem to admit a nice separation of proof from checking.

Then if a proof is required, there seems no significantly easier way than generating proofs along each step of the algorithm.

Example: Cohen-Hörmander algorithm implemented in HOL Light by McLaughlin (CADE 2005).

Works well, useful for small problems, but about $1000\times$ slowdown relative to non-proof-producing implementation.

Should we use reflection, i.e. verify the code itself?

Summary

- There is a need for combinations of decision methods
- For general quantifier prefixes, relatively few useful results.
- Nelson-Oppen and Shostak give useful methods for universal formulas.
- We sometimes also want decision procedures to produce proofs
- Some procedures admit efficient separation of search and checking, others do not.
- Interesting research topic: new ways of compactly certifying decision methods.

Theorem Proving for Verification
4(a): Cohen-Hörmander real quantifier elimination

John Harrison

Intel Corporation

Marktoberdorf 2010

Sat 14th August 2010 (08:45 – 09:30)

Summary

- What we'll prove
- History
- Sign matrices
- The key recursion
- Parametrization
- Real-closed fields

What we'll prove

Take a first-order language:

- All rational constants p/q
- Operators of negation, addition, subtraction and multiplication
- Relations '$=$', '$<$', '$\leqslant$', '$>$', '$\geqslant$'

We'll prove that every formula in the language has a quantifier-free equivalent, and will give a systematic algorithm for finding it.

Applications

In principle, this method can be used to solve many non-trivial problems.

> Kissing problem: how many disjoint n-dimensional spheres can be packed into space so that they touch a given unit sphere?

Pretty much *any* geometrical assertion can be expressed in this theory.

If theorem holds for *complex* values of the coordinates, and then simpler methods are available (Gröbner bases, Wu-Ritt triangulation. . .).

History

- 1930: Tarski discovers quantifier elimination procedure for this theory.
- 1948: Tarski's algorithm published by RAND
- 1954: Seidenberg publishes simpler algorithm
- 1975: Collins develops and *implements* cylindrical algebraic decomposition (CAD) algorithm
- 1983: Hörmander publishes very simple algorithm based on ideas by Cohen.
- 1990: Vorobjov improves complexity bound to doubly exponential in number of quantifier *alternations*.

We'll present the Cohen-Hörmander algorithm.

Current implementations

There are quite a few simple versions of real quantifier elimination, even in computer algebra systems like Mathematica.

Among the more heavyweight implementations are:

- `qepcad` — `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`
- `REDLOG` — `http://www.fmi.uni-passau.de/~redlog/`

One quantifier at a time

For a general quantifier elimination procedure, we just need one for a formula

$$\exists x.\ P[a_1, \ldots, a_n, x]$$

where $P[a_1, \ldots, a_n, x]$ involves no other quantifiers but may involve other variables.

Then we can apply the procedure successively inside to outside, dealing with universal quantifiers via $(\forall x.\ P[x]) \Leftrightarrow (\neg\exists x.\ \neg P[x])$.

Forget parametrization for now

First we'll ignore the fact that the polynomials contain variables other than the one being eliminated.

This keeps the technicalities a bit simpler and shows the main ideas clearly.

The generalization to the parametrized case will then be very easy:

- Replace polynomial division by pseudo-division
- Perform case-splits to determine signs of coefficients

Sign matrices

Take a set of univariate polynomials $p_1(x), \ldots, p_n(x)$.

A *sign matrix* for those polynomials is a division of the real line into alternating points and intervals:

$$(-\infty, x_1), x_1, (x_1, x_2), x_2, \ldots, x_{m-1}, (x_{m-1}, x_m), x_m, (x_m, +\infty)$$

and a matrix giving the sign of each polynomial on each interval:

- Positive ($+$)
- Negative ($-$)
- Zero (0)

Sign matrix example

The polynomials $p_1(x) = x^2 - 3x + 2$ and $p_2(x) = 2x - 3$ have the following sign matrix:

Point/Interval	p_1	p_2
$(-\infty, x_1)$	$+$	$-$
x_1	0	$-$
(x_1, x_2)	$-$	$-$
x_2	$-$	0
(x_2, x_3)	$-$	$+$
x_3	0	$+$
$(x_3, +\infty)$	$+$	$+$

Using the sign matrix

Using the sign matrix for all polynomials appearing in $P[x]$ we can answer any quantifier elimination problem: $\exists x.\ P[x]$

- Look to see if any row of the matrix satisfies the formula (hence dealing with existential)
- For each row, just see if the corresponding set of signs satisfies the formula.

We have replaced the quantifier elimination problem with sign matrix determination

Finding the sign matrix

For constant polynomials, the sign matrix is trivial (2 has sign '$+$' etc.)

To find a sign matrix for $p, p_1, \ldots, p_n$ it suffices to find one for $p', p_1, \ldots, p_n, r_0, r_1, \ldots, r_n$, where

- $p_0 \equiv p'$ is the derivative of p
- $r_i = \text{rem}(p, p_i)$

(Remaindering means we have some q_i so $p = q_i \cdot p_i + r_i$.)

Taking p to be the polynomial of highest degree we get a simple recursive algorithm for sign matrix determination.

Details of recursive step

So, suppose we have a sign matrix for $p', p_1, \ldots, p_n, r_0, r_1, \ldots, r_n$.

We need to construct a sign matrix for $p, p_1, \ldots, p_n$.

- May need to add more points and hence intervals for roots of p
- Need to determine signs of $p_1, \ldots, p_n$ at the new points and intervals
- Need the sign of p itself everywhere.

Step 1

Split the given sign matrix into two parts, but keep all the points for now:

- M for $p', p_1, \ldots, p_n$
- M' for $r_0, r_1, \ldots, r_n$

We can infer the sign of p at all the 'significant' *points* of M as follows:

$$p = q_i p_i + r_i$$

and for each of our points, one of the p_i is zero, so $p = r_i$ there and we can read off p's sign from r_i's.

Step 2

Now we're done with M' and we can throw it away.

We also 'condense' M by eliminating points that are not roots of one of the $p', p_1, \ldots, p_n$.

Note that the sign of any of these polynomials is stable on the condensed intervals, since they have no roots there.

- We know the sign of p at all the points of this matrix.
- However, p itself may have additional roots, and we don't know anything about the intervals yet.

Step 3

There can be at most one root of p in each of the existing intervals, because otherwise p' would have a root there.

We can tell whether there is a root by checking the signs of p (determined in Step 1) at the two endpoints of the interval.

Insert a new point precisely if p has strictly opposite signs at the two endpoints (simple variant for the two end intervals).

None of the other polynomials change sign over the original interval, so just copy the values to the point and subintervals.

Throw away p' and we're done!

Multivariate generalization

In the multivariate context, we can't simply divide polynomials. Instead of

$$p = p_i \cdot q_i + r_i$$

we get

$$a^k p = p_i \cdot q_i + r_i$$

where a is the leading coefficient of p_i.

The same logic works, but we need case splits to fix the sign of a.

Real-closed fields

With more effort, all the ‘analytical’ facts can be deduced from the axioms for *real-closed fields*.

- Usual ordered field axioms
- Existence of square roots: $\forall x.\ x \geqslant 0 \Rightarrow \exists y.\ x = y^2$
- Solvability of odd-degree equations:
  $\forall a_0, \ldots, a_n.\ a_n \neq 0 \Rightarrow \exists x.\ a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$

Examples include computable reals and algebraic reals. So this already gives a complete theory, without a stronger completeness axiom.

Summary

- Real quantifier elimination one of the most significant logical decidability results known.
- Original result due to Tarski, for general real closed fields.
- A half-century of research has resulted in simpler and more efficient algorithms (not always at the same time).
- The Cohen-Hörmander algorithm is remarkably simple (relatively speaking).
- The complexity, both theoretical and practical, is still bad, so there's limited success on non-trivial problems.

Theorem Proving for Verification
4(b): Interactive theorem proving

John Harrison
Intel Corporation

Marktoberdorf 2010

Sat 14th August 2010 (08:45 – 09:30)

Interactive theorem proving (1)

In practice, many interesting problems can't be automated completely:

- They don't fall in a practical decidable subset
- Pure first order proof search is not a feasible approach with, e.g. set theory

Interactive theorem proving (1)

In practice, most interesting problems can't be automated completely:

- They don't fall in a practical decidable subset
- Pure first order proof search is not a feasible approach with, e.g. set theory

In practice, we need an interactive arrangement, where the user and machine work together.

The user can delegate simple subtasks to pure first order proof search or one of the decidable subsets.

However, at the high level, the user must guide the prover.

Interactive theorem proving (2)

The idea of a more ‘interactive’ approach was already anticipated by pioneers, e.g. Wang (1960):

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous that *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

However, constructing an effective and programmable combination is not so easy.

SAM

First successful family of interactive provers were the SAM systems:

> Semi-automated mathematics is an approach to theorem-proving which seeks to combine automatic logic routines with ordinary proof procedures in such a manner that the resulting procedure is both efficient and subject to human intervention in the form of control and guidance. Because it makes the mathematician an essential factor in the quest to establish theorems, this approach is a departure from the usual theorem-proving attempts in which the computer *unaided* seeks to establish proofs.

SAM V was used to settle an open problem in lattice theory.

Three influential proof checkers

- AUTOMATH (de Bruijn, . . .) — Implementation of type theory, used to check non-trivial mathematics such as Landau's *Grundlagen*
- Mizar (Trybulec, . . .) — Block-structured natural deduction with 'declarative' justifications, used to formalize large body of mathematics
- LCF (Milner et al) — Programmable proof checker for Scott's Logic of Computable Functions written in new functional language ML.

Ideas from all these systems are used in present-day systems. (Corbineau's declarative proof mode for Coq . . .)

Sound extensibility

Ideally, it should be possible to customize and program the theorem-prover with domain-specific proof procedures.

However, it's difficult to allow this without compromising the soundness of the system.

A very successful way to combine extensibility and reliability was pioneered in LCF.

Now used in Coq, HOL, Isabelle, Nuprl, ProofPower, ...

Key ideas behind LCF

- Implement in a strongly-typed functional programming language (usually a variant of ML)
- Make `thm` (‘theorem’) an abstract data type with only simple primitive inference rules
- Make the implementation language available for arbitrary extensions.

First-order axioms (1)

$$\vdash \quad p \Rightarrow (q \Rightarrow p)$$

$$\vdash \quad (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow (p \Rightarrow r)$$

$$\vdash \quad ((p \Rightarrow \bot) \Rightarrow \bot) \Rightarrow p$$

$$\vdash \quad (\forall x.\, p \Rightarrow q) \Rightarrow (\forall x.\, p) \Rightarrow (\forall x.\, q)$$

$\vdash \quad p \Rightarrow \forall x.\, p$ **[Provided** $x \notin \mathsf{FV}(p)$**]**

$\vdash \quad (\exists x.\, x = t)$ **[Provided** $x \notin \mathsf{FVT}(t)$**]**

$$\vdash \quad t = t$$

$$\vdash \quad s_1 = t_1 \Rightarrow ... \Rightarrow s_n = t_n \Rightarrow f(s_1, .., s_n) = f(t_1, .., t_n)$$

$$\vdash \quad s_1 = t_1 \Rightarrow ... \Rightarrow s_n = t_n \Rightarrow P(s_1, .., s_n) \Rightarrow P(t_1, .., t_n)$$

First-order axioms (2)

$$\vdash \quad (p \Leftrightarrow q) \Rightarrow p \Rightarrow q$$

$$\vdash \quad (p \Leftrightarrow q) \Rightarrow q \Rightarrow p$$

$$\vdash \quad (p \Rightarrow q) \Rightarrow (q \Rightarrow p) \Rightarrow (p \Leftrightarrow q)$$

$$\vdash \quad \top \Leftrightarrow (\bot \Rightarrow \bot)$$

$$\vdash \quad \neg p \Leftrightarrow (p \Rightarrow \bot)$$

$$\vdash \quad p \wedge q \Leftrightarrow (p \Rightarrow q \Rightarrow \bot) \Rightarrow \bot$$

$$\vdash \quad p \vee q \Leftrightarrow \neg(\neg p \wedge \neg q)$$

$$\vdash \quad (\exists x.\, p) \Leftrightarrow \neg(\forall x.\, \neg p)$$

First-order rules

Modus Ponens rule:

$$\frac{\vdash p \Rightarrow q \qquad \vdash p}{\vdash q}$$

Generalization rule:

$$\frac{\vdash p}{\vdash \forall x.\, p}$$

LCF kernel for first order logic (1)

Define type of first order formulas:

```
type term = Var of string | Fn of string * term list;;

type formula = False
             | True
             | Atom of string * term list
             | Not of formula
             | And of formula * formula
             | Or of formula * formula
             | Imp of formula * formula
             | Iff of formula * formula
             | Forall of string * formula
             | Exists of string * formula;;
```

LCF kernel for first order logic (2)

Define some useful helper functions:

```
let mk_eq s t = Atom(R("=",[s;t]));;

let rec occurs_in s t =
  s = t or
  match t with
    Var y -> false
  | Fn(f,args) -> exists (occurs_in s) args;;

let rec free_in t fm =
  match fm with
    False | True -> false
  | Atom(R(p,args)) -> exists (occurs_in t) args
  | Not(p) -> free_in t p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> free_in t p or free_in t q
  | Forall(y,p) | Exists(y,p) -> not(occurs_in (Var y) t) & free_in t p;;
```

LCF kernel for first order logic (3)

```
module Proven : Proofsystem =
  struct type thm = formula
         let axiom_addimp p q = Imp(p,Imp(q,p))
         let axiom_distribimp p q r = Imp(Imp(p,Imp(q,r)),Imp(Imp(p,q),Imp(p,r)))
         let axiom_doubleneg p = Imp(Imp(Imp(p,False),False),p)
         let axiom_allimp x p q = Imp(Forall(x,Imp(p,q)),Imp(Forall(x,p),Forall(x,q)))
         let axiom_impall x p =
           if not (free_in (Var x) p) then Imp(p,Forall(x,p)) else failwith "axiom_impall"
         let axiom_existseq x t =
           if not (occurs_in (Var x) t) then Exists(x,mk_eq (Var x) t) else failwith "axiom_existseq"
         let axiom_eqrefl t = mk_eq t t
         let axiom_funcong f lefts rights =
            itlist2 (fun s t p -> Imp(mk_eq s t,p)) lefts rights (mk_eq (Fn(f,lefts)) (Fn(f,rights)))
         let axiom_predcong p lefts rights =
            itlist2 (fun s t p -> Imp(mk_eq s t,p)) lefts rights (Imp(Atom(p,lefts),Atom(p,rights)))
         let axiom_iffimp1 p q = Imp(Iff(p,q),Imp(p,q))
         let axiom_iffimp2 p q = Imp(Iff(p,q),Imp(q,p))
         let axiom_impiff p q = Imp(Imp(p,q),Imp(Imp(q,p),Iff(p,q)))
         let axiom_true = Iff(True,Imp(False,False))
         let axiom_not p = Iff(Not p,Imp(p,False))
         let axiom_or p q = Iff(Or(p,q),Not(And(Not(p),Not(q))))
         let axiom_and p q = Iff(And(p,q),Imp(Imp(p,Imp(q,False)),False))
         let axiom_exists x p = Iff(Exists(x,p),Not(Forall(x,Not p)))
         let modusponens pq p =
           match pq with Imp(p',q) when p = p' -> q | _ -> failwith "modusponens"
         let gen x p = Forall(x,p)
         let concl c = c
  end;;
```

Derived rules

The primitive rules are very simple. But using the LCF technique we can build up a set of derived rules. The following derives $p \Rightarrow p$:

```
let imp_refl p = modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
                                          (axiom_addimp p (Imp(p,p))))
                             (axiom_addimp p p);;
```

Derived rules

The primitive rules are very simple. But using the LCF technique we can build up a set of derived rules. The following derives $p \Rightarrow p$:

```
let imp_refl p = modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
                                          (axiom_addimp p (Imp(p,p))))
                             (axiom_addimp p p);;
```

While this process is tedious at the beginning, we can quickly reach the stage of automatic derived rules that

- Prove propositional tautologies
- Perform Knuth-Bendix completion
- Prove first order formulas by standard proof search and translation

Fully-expansive decision procedures

Real LCF-style theorem provers like HOL have many powerful derived rules.

Mostly just mimic standard algorithms like rewriting but by inference. For cases where this is difficult:

- Separate certification (my previous lecture)
- Reflection

Proof styles

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable

Mizar pioneered the declarative style of proof.

Recently, several other declarative proof languages have been developed, as well as declarative shells round existing systems like HOL and Isabelle.

Finding the right style is an interesting research topic.

Procedural proof example

```
let NSQRT_2 = prove
 (`!p q. p * p = 2 * q * q ==> q = 0`,
  MATCH_MP_TAC num_WF THEN REWRITE_TAC[RIGHT_IMP_FORALL_THM] THEN
  REPEAT STRIP_TAC THEN FIRST_ASSUM(MP_TAC o AP_TERM `EVEN`) THEN
  REWRITE_TAC[EVEN_MULT; ARITH] THEN REWRITE_TAC[EVEN_EXISTS] THEN
  DISCH_THEN(X_CHOOSE_THEN `m:num` SUBST_ALL_TAC) THEN
  FIRST_X_ASSUM(MP_TAC o SPECL [`q:num`; `m:num`]) THEN
  ASM_REWRITE_TAC[ARITH_RULE
   `q < 2 * m ==> q * q = 2 * m * m ==> m = 0 <=>
    (2 * m) * 2 * m = 2 * q * q ==> 2 * m <= q`] THEN
  ASM_MESON_TAC[LE_MULT2; MULT_EQ_0; ARITH_RULE `2 * x <= x <=> x = 0`]);;
```

Declarative proof example

```
let NSQRT_2 = prove
 (`!p q. p * p = 2 * q * q ==> q = 0`,
  suffices_to_prove
   `!p. (!m. m < p ==> (!q. m * m = 2 * q * q ==> q = 0))
        ==> (!q. p * p = 2 * q * q ==> q = 0)`
   (wellfounded_induction) THEN
  fix [`p:num`] THEN
  assume("A") `!m. m < p ==> !q. m * m = 2 * q * q ==> q = 0` THEN
  fix [`q:num`] THEN
  assume("B") `p * p = 2 * q * q` THEN
  so have `EVEN(p * p) <=> EVEN(2 * q * q)` (trivial) THEN
  so have `EVEN(p)` (using [ARITH; EVEN_MULT] trivial) THEN
  so consider (`m:num`,"C",`p = 2 * m`) (using [EVEN_EXISTS] trivial) THEN
  cases ("D",`q < p \/ p <= q`) (arithmetic) THENL
   [so have `q * q = 2 * m * m ==> m = 0` (by ["A"] trivial) THEN
    so we're finished (by ["B"; "C"] algebra);
    so have `p * p <= q * q` (using [LE_MULT2] trivial) THEN
    so have `q * q = 0` (by ["B"] arithmetic) THEN
    so we're finished (algebra)]);;
```

Is automation even more declarative?

```
let LEMMA_1 = SOS_RULE
 `p EXP 2 = 2 * q EXP 2
  ==> (q = 0 \/ 2 * q - p < p /\ ~(p - q = 0)) /\
      (2 * q - p) EXP 2 = 2 * (p - q) EXP 2`;;

let NSQRT_2 = prove
 (`!p q. p * p = 2 * q * q ==> q = 0`,
  REWRITE_TAC[GSYM EXP_2] THEN MATCH_MP_TAC num_WF THEN MESON_TAC[LEMMA_1]);;
```

The Seventeen Provers of the World (1)

- ACL2 — Highly automated prover for first-order number theory without explicit quantifiers, able to do induction proofs itself.
- Alfa/Agda — Prover for constructive type theory integrated with dependently typed programming language.
- B prover — Prover for first-order set theory designed to support verification and refinement of programs.
- Coq — LCF-like prover for constructive Calculus of Constructions with reflective programming language.
- HOL (HOL Light, HOL4, ProofPower) — Seminal LCF-style prover for classical simply typed higher-order logic.
- IMPS — Interactive prover for an expressive logic supporting partially defined functions.

The Seventeen Provers of the World (2)

- Isabelle/Isar — Generic prover in LCF style with a newer declarative proof style influenced by Mizar.
- Lego — Well-established framework for proof in constructive type theory, with a similar logic to Coq.
- Metamath — Fast proof checker for an exceptionally simple axiomatization of standard ZF set theory.
- Minlog — Prover for minimal logic supporting practical extraction of programs from proofs.
- Mizar — Pioneering system for formalizing mathematics, originating the declarative style of proof.
- Nuprl/MetaPRL — LCF-style prover with powerful graphical interface for Martin-Löf type theory with new constructs.

The Seventeen Provers of the World (3)

- Omega — Unified combination in modular style of several theorem-proving techniques including proof planning.
- Otter/IVY — Powerful automated theorem prover for pure first-order logic plus a proof checker.
- PVS — Prover designed for applications with an expressive classical type theory and powerful automation.
- PhoX — prover for higher-order logic designed to be relatively simple to use in comparison with Coq, HOL etc.
- Theorema — Ambitious integrated framework for theorem proving and computer algebra built inside Mathematica.

For more, see Freek Wiedijk, *The Seventeen Provers of the World*, Springer Lecture Notes in Computer Science vol. 3600, 2006.

Summary

- In practice, we need a combination of interaction and automation for difficult proofs.
- Interactive provers / proof checkers are the workhorses in verification applications, even if they use automated subsystems.
- LCF gives a good way of realizing a combination of soundness and extensibility.
- Different proof styles may be preferable, and they can be supported on top of an LCF-style core.
- There are many interactive provers out there with very different characteristics!