A Survey of Automated Theorem Proving

John Harrison

Intel Corporation

28–29 September 2013

1: Background, history and propositional logic

John Harrison, Intel Corporation
Computer Science Club, St. Petersburg
Sat 28th September 2013 (17:20–18:55)

What I will talk about

Aim is to cover some of the most important approaches to computer-aided proof in classical logic.

This is usually called 'automated theorem proving' or 'automated reasoning', though we interpret "automated" quite broadly.

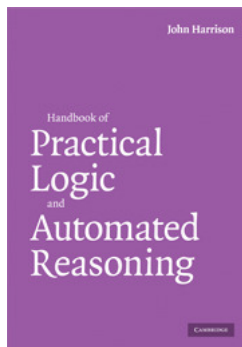
1. Background and propositional logic
2. First-order logic, with and without equality
3. Decidable problems in logic and algebra
4. Interactive theorem proving
5. Applications to mathematics and computer verification

What I won't talk about

- Temporal logic, model checking etc.
- Higher-order logic and type theory
- Constructive logic, modal logic, other nonclassical logics

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`



What is automated reasoning?

Attempting to perform logical reasoning in an automatic and algorithmic way. An old dream:

- Hobbes (1651): "*Reason* . . . is nothing but *reckoning* (that is, adding and subtracting) of the consequences of general names agreed upon, for the *marking* and *signifying* of our thoughts."
- Leibniz (1685) "When there are disputes among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right."

Nowadays, by 'automatic and algorithmic' we mean 'using a computer program'.

What does automated reasoning involve?

There are two steps to performing automated reasoning, as anticipated by Leibniz:

- Express *statement* of theorems in a formal language. (Leibniz's *characteristica universalis*.)
- Use automated algorithmic manipulations on those formal expressions. (Leibniz's *calculus ratiocinator*).

Is that really possible?

Theoretical and practical limitations

- Limitative results in logic (Gödel, Tarski, Church-Turing, Matiyasevich) imply that not even elementary number theory can be done completely automatically.
- There *are* formal proof systems (e.g. first-order set theory) and semi-decision procedures that will in principle find the proof of anything provable in 'ordinary' mathematics.
- In practice, because of time or space limits, these automated procedures are not all that useful, and we may prefer an interactive arrangement where a human guides the machine.

Why automated reasoning?

For general intellectual interest? It is a fascinating field that helps to understand the real nature of mathematical creativity. Or more practically:

- To check the correctness of proofs in mathematics, supplementing or even replacing the existing 'social process' of peer review etc. with a more objective criterion.
- To extend rigorous proof from pure mathematics to the verification of computer systems (programs, hardware systems, protocols etc.), supplementing or replacing the usual testing process.

These are currently the two main drivers of progress in the field.

Theorem provers vs. computer algebra systems

Both systems for symbolic computation, but rather different:

- Theorem provers are more logically flexible and rigorous
- CASs are generally easier to use and more efficient/powerful

Some systems like MathXpert, Theorema blur the distinction somewhat . . .

Limited expressivity in CASs

Often limited to conditional equations like

$$\sqrt{x^2} = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x \leq 0 \end{cases}$$

whereas using logic we can say many interesting (and highly undecidable) things

$$\forall x \in \mathbb{R}.\, \forall \epsilon > 0.\, \exists \delta > 0.\, \forall x'.\, |x - x'| < \delta \Rightarrow |f(x) - f(x')| < \epsilon$$

Unclear expressions in CASs

Consider an equation $(x^2 - 1)/(x - 1) = x + 1$ from a CAS. What does it mean?

- Universally valid identity (albeit not quite valid)?
- Identity true when both sides are defined
- Identity over the field of rational functions
- ...

Lack of rigour in many CASs

CASs often apply simplifications even when they are not strictly valid.
Hence they can return wrong results.
Consider the evaluation of this integral in Maple:

$$\int_0^\infty \frac{e^{-(x-1)^2}}{\sqrt{x}} dx$$

We try it two different ways:

An integral in Maple

```
> int(exp(-(x-t)^2)/sqrt(x), x=0..infinity);
```

$$\frac{1}{2}\frac{e^{-t^2}\left(-\frac{3(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{3}{4}}(\frac{t^2}{2})}{t^2}+(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{7}{4}}(\frac{t^2}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> subs(t=1,%);
```

$$\frac{1}{2}\frac{e^{-1}\left(-3\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{3}{4}}(\frac{1}{2})+\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{7}{4}}(\frac{1}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> evalf(%);
```

0.4118623312

```
> evalf(int(exp(-(x-1)^2)/sqrt(x), x=0..infinity));
```

1.973732150

Early research in automated reasoning

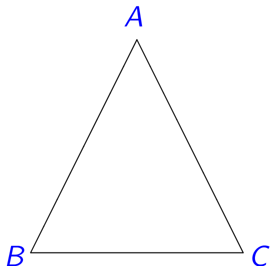
Most early theorem provers were fully automatic, even though there were several different approaches:

- Human-oriented AI style approaches (Newell-Simon, Gelerntner)
- Machine-oriented algorithmic approaches (Davis, Gilmore, Wang, Prawitz)

Modern work dominated by machine-oriented approach but some successes for AI approach.

A theorem in geometry (1)

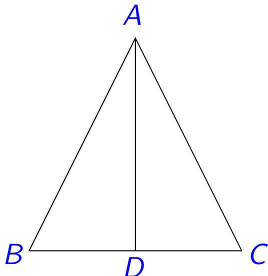
Example of AI approach in action:



If the sides AB and AC are equal (i.e. the triangle is isoseles), then the angles ABC and ACB are equal.

A theorem in geometry (2)

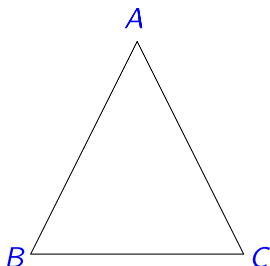
Drop perpendicular meeting BC at a point D:



and then use the fact that the triangles ABD and ACD are congruent.

A theorem in geometry (3)

Originally found by Pappus but not in many books:



Simply, the triangles ABC and ACB are congruent.

The Robbins Conjecture (1)

Huntington (1933) presented the following axioms for a Boolean algebra:

$$\begin{aligned} x + y &= y + x \\ (x + y) + z &= x + (y + z) \\ n(n(x) + y) + n(n(x) + n(y)) &= x \end{aligned}$$

Herbert Robbins conjectured that the Huntington equation can be replaced by a simpler one:

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

The Robbins Conjecture (2)

This conjecture went unproved for more than 50 years, despite being studied by many mathematicians, even including Tarski.

It because a popular target for researchers in automated reasoning.

In October 1996, a (key lemma leading to) a proof was found by McCune's program EQP.

The successful search took about 8 days on an RS/6000 processor and used about 30 megabytes of memory.

What can be automated?

- Validity/satisfiability in propositional logic is decidable (SAT).
- Validity/satisfiability in many temporal logics is decidable.
- Validity in first-order logic is *semidecidable*, i.e. there are complete proof procedures that may run forever on invalid formulas
- Validity in higher-order logic is not even *semidecidable* (or anywhere in the arithmetical hierarchy).

Some specific theories

We are often interested in validity w.r.t. some suitable background theory.

- Linear theory of $\mathbb{N}$ or $\mathbb{Z}$ is decidable. Nonlinear theory not even semidecidable.
- Linear and nonlinear theory of $\mathbb{R}$ is decidable, though complexity is very bad in the nonlinear case.
- Linear and nonlinear theory of $\mathbb{C}$ is decidable. Commonly used in geometry.

Many of these naturally generalize known algorithms like linear/integer programming and Sturm's theorem.

Propositional Logic

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	$p \Rightarrow q$	$p \leq q$	$p \rightarrow q$, $p \supset q$
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.

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

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':

$$
\begin{array}{rcl}
p \vee (q \wedge r) & \mapsto & (p \vee q) \wedge (p \vee r) \\
(p \wedge q) \vee r & \mapsto & (p \vee r) \wedge (q \vee r)
\end{array}
$$

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':

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

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

$$\begin{array}{l}(\neg p_1 \lor q) \land (\neg p_1 \lor \neg r) \land (p_1 \lor \neg q \lor r) \land \\ (\neg p_2 \lor p \lor p_1) \land (p_2 \lor \neg p) \land (p_2 \lor \neg p_1) \land \\ (\neg p_3 \lor p_2) \land (\neg p_3 \lor s) \land (p_3 \lor \neg p_2 \lor \neg s) \land p_3\end{array}$$

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:

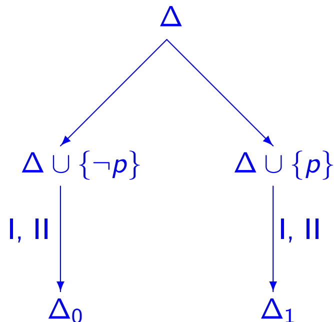
I The 1-literal rule: if a unit clause p appears, remove $\neg p$ from other clauses and remove all clauses including p.

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

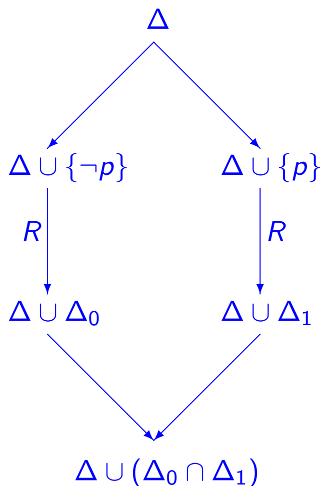
Suppose we have clauses

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

If we split over variables in the order $p_1,\ldots,p_{10}$, assuming first that they are true, we then get a conflict.
Yet none of the assignments to $p_2,\ldots,p_9$ 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.

Stålmarck's algorithm

Stålmarck's 'dilemma' rule attempts to avoid nested case splits by feeding back common information from both branches.



2: First-order logic with and without equality

John Harrison, Intel Corporation
Computer Science Club, St. Petersburg
Sat 28th September 2013 (19:05–20:40)

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

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

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

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

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

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

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.

$$\begin{array}{l} \forall x_1, \ldots, x_n.\, \exists y.\, P[x_1, \ldots, x_n, y] \\ \to \forall x_1, \ldots, x_n.\, P[x_1, \ldots, x_n, f(x_1, \ldots, x_n)] \end{array}$$

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* and *compactness theorem*, together implying:

> *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 all finite sets 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:

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

Instantiation versus 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.
Humans tend to find instantiations intelligently based on some understanding of the problem.
Even for the machine, instantiations can be chosen more intelligently by a syntax-driven process of *unification*.
For example, choose instantiation for x and y so that $D(x)$ and $\neg(D(f(y)))$ are complementary.

Unification

Given a set of pairs of terms

$$S = \{(s_1, t_1), \ldots, (s_n, t_n)\}$$

a *unifier* of S is an instantiation σ such that each

$$\sigma s_i = \sigma t_i$$

If a unifier exists there is a *most general* unifier (MGU), of which any other is an instance.
MGUs can be found by straightforward recursive algorithm.

Unification-based theorem proving

Many theorem-proving algorithms based on unification exist:

- Tableaux
- Resolution / inverse method / superposition
- Model elimination
- Connection method
- ...

Unification-based theorem proving

Many theorem-proving algorithms based on unification exist:

- Tableaux
- Resolution / inverse method / superposition
- Model elimination
- Connection method
- ...

Roughly, you can take a propositional decision procedure and "lift" it to a first-order one by adding unification, though there are subtleties:

- Distinction between top-down and bottom-up methods
- Need for factoring in resolution

Resolution

Propositional resolution is the rule:

$$\frac{p \vee A \quad \neg p \vee B}{A \vee B}$$

and full first-order resolution is the generalization

$$\frac{P \vee A \quad Q \vee B}{\sigma(A \vee B)}$$

where σ is an MGU of literal sets P and Q^{-}.

Factoring

Pure propositional resolution is (refutation) complete in itself, but in the first-order context we may in general need ‘factoring’.

Idea: we may need to make an instantiation more special to collapse a set of literals into a smaller set.

Example: there does not exist a barber who shaves exactly the people who do not shave themselves:

$$\exists b. \forall x. \mathit{shaves}(b, x) \Leftrightarrow \neg \mathit{shaves}(x, x)$$

If we reduce to clauses we get the following to refute:

$$\{\neg \mathit{shaves}(x, x) \lor \neg \mathit{shaves}(b, x)\}, \{\mathit{shaves}(x, x) \lor \mathit{shaves}(b, x)\}$$

and resolution doesn’t derive useful consequences without factoring.

Adding equality

We often want to restrict ourselves to validity in *normal* models where 'equality means equality'.

- Add extra axioms for equality and use non-equality decision procedures
- Use other preprocessing methods such as Brand transformation or STE
- Use special rules for equality such as paramodulation or superposition

Equality axioms

Given a formula p, let the *equality axioms* be equivalence:

$$\forall x.\, x = x$$
$$\forall x\ y.\, x = y \Rightarrow y = x$$
$$\forall x\ y\ z.\, x = y \land y = z \Rightarrow x = z$$

together with *congruence* rules for each function and predicate in p:

$$\forall \overline{xy}.\, x_1 = y_1 \land \cdots \land x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$
$$\forall \overline{xy}.\, x_1 = y_1 \land \cdots \land x_n = y_n \Rightarrow R(x_1, \ldots, x_n) \Rightarrow R(y_1, \ldots, y_n)$$

Brand transformation

Adding equality axioms has a bad reputation in the ATP world.
Simple substitutions like $x = y \Rightarrow f(y) + f(f(x)) = f(x) + f(f(y))$ need many applications of the rules.
Brand's transformation uses a different translation to build in equality, involving 'flattening'

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$x \cdot y = w_1 \Rightarrow w_1 \cdot z = x \cdot (y \cdot z)$$
$$x \cdot y = w_1 \land y \cdot z = w_2 \Rightarrow w_1 \cdot z = x \cdot w_2$$

Still not conclusively better.

Paramodulation and related methods

Often better to add special rules such as paramodulation:

$$\frac{C \vee s \doteq t \qquad D \vee P[s']}{\sigma\,(C \vee D \vee P[t])}$$

Works best with several restrictions including the use of orderings to orient equations.

Easier to understand for pure equational logic.

Normalization by rewriting

Use a set of equations left-to-right as rewrite rules to simplify or normalize a term:

- Use some kind of ordering (e.g. lexicographic path order) to ensure termination
- Difficulty is ensuring confluence

Failure of confluence

Consider these axioms for groups:

$$
\begin{aligned}
&(x \cdot y) \cdot z = x \cdot (y \cdot z) \\
&1 \cdot x = x \\
&i(x) \cdot x = 1
\end{aligned}
$$

They are not confluent because we can rewrite

$$
\begin{aligned}
(i(x) \cdot x) \cdot y &\longrightarrow i(x) \cdot (x \cdot y) \\
(i(x) \cdot x) \cdot y &\longrightarrow 1 \cdot y
\end{aligned}
$$

Knuth-Bendix completion

Key ideas of Knuth-Bendix completion:

- Use unification to identify most general situations where confluence fails ('critical pairs')
- Add critical pairs, suitably oriented, as new equations and repeat

This process completes the group axioms, deducing some non-trivial consequences along the way.

Completion of group axioms

$$
\begin{aligned}
&i(x \cdot y) = i(y) \cdot i(x)\\
&i(i(x)) = x\\
&i(1) = 1\\
&x \cdot i(x) = 1\\
&x \cdot i(x) \cdot y = y\\
&x \cdot 1 = x\\
&i(x) \cdot x \cdot y = y\\
&1 \cdot x = x\\
&i(x) \cdot x = 1\\
&(x \cdot y) \cdot z = x \cdot y \cdot z
\end{aligned}
$$

Decidable fragments of F.O.L.

Validity in first-order logic is only semidecidable (Church-Turing). However, there are some interesting special cases where it is decidable, e.g.

Decidable fragments of F.O.L.

Validity in first-order logic is only semidecidable (Church-Turing). However, there are some interesting special cases where it is decidable, e.g.

- AE formulas: no function symbols, universal quantifiers before existentials in prenex form
- Monadic formulas: no function symbols, only unary predicates

Decidable fragments of F.O.L.

Validity in first-order logic is only semidecidable (Church-Turing). However, there are some interesting special cases where it is decidable, e.g.

- AE formulas: no function symbols, universal quantifiers before existentials in prenex form
- Monadic formulas: no function symbols, only unary predicates

All 'syllogistic' reasoning can be reduced to the monadic fragment:

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

can be expressed as the monadic formula:

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

Why AE is decidable

The negation of an AE formula is an EA formula to be refuted:

$$\exists x_1, \ldots, x_n.\, \forall y_1, \ldots, y_m.\, P[x_1, \ldots, x_n, y_1, \ldots, y_m]$$

and after Skolemization we still have no functions:

$$\forall y_1, \ldots, y_m.\, P[c_1, \ldots, c_n, y_1, \ldots, y_m]$$

So there are only finitely many ground instances to check for satisfiability.

Why AE is decidable

The negation of an AE formula is an EA formula to be refuted:

$$\exists x_1, \ldots, x_n. \forall y_1, \ldots, y_m. P[x_1, \ldots, x_n, y_1, \ldots, y_m]$$

and after Skolemization we still have no functions:

$$\forall y_1, \ldots, y_m. P[c_1, \ldots, c_n, y_1, \ldots, y_m]$$

So there are only finitely many ground instances to check for satisfiability.
Since the equality axioms are purely universal formulas, adding those doesn't disturb the AE/EA nature, so we get Ramsey's decidability result.

The finite model property

Another way of understanding decidability results is that fragments like AE and monadic formulas have the finite model property:

> *If the formula in the fragment has a model it has a finite model.*

Any fragment with the finite model property is decidable: search for a model and a disproof in parallel.
Often we even know the exact size we need consider: e.g. size 2^n for monadic formula with n predicates.
In practice, we quite often find finite countermodels to false formulas.

Failures of the FMP

However many formulas with simple quantifier prefixes *don't* have the FMP:

- $(\forall x.\ \neg R(x,x)) \land (\forall x.\ \exists z.\ R(x,z)) \land$
  $(\forall x\ y\ z.\ R(x,y) \land R(y,z) \Rightarrow R(x,z))$
- $(\forall x.\ \neg R(x,x)) \land (\forall x.\ \exists y.\ R(x,y) \land \forall z.\ R(y,z) \Rightarrow R(x,z)))$
- $\neg(\ (\forall x.\ \neg(F(x,x)) \land$
  $(\forall x\ y.\ F(x,y) \Rightarrow F(y,x)) \land$
  $(\forall x\ y.\ \neg(x = y) \Rightarrow \exists! z.\ \ F(x,z) \land F(y,z))$
  $\Rightarrow \exists u.\ \forall v.\ \neg(v = u) \Rightarrow F(u,v))$

Failures of the FMP

However many formulas with simple quantifier prefixes *don't* have the FMP:

- $(\forall x.\, \neg R(x,x)) \wedge (\forall x.\, \exists z.\, R(x,z)) \wedge$
  $(\forall x\ y\ z.\, R(x,y) \wedge R(y,z) \Rightarrow R(x,z))$
- $(\forall x.\, \neg R(x,x)) \wedge (\forall x.\, \exists y.\, R(x,y) \wedge \forall z.\, R(y,z) \Rightarrow R(x,z)))$
- $$\begin{array}{ll} \neg(& (\forall x.\, \neg(F(x,x)) \wedge \\ & (\forall x\ y.\, F(x,y) \Rightarrow F(y,x)) \wedge \\ & (\forall x\ y.\, \neg(x = y) \Rightarrow \exists z.\ \ F(x,z) \wedge F(y,z) \wedge \\ & \qquad\qquad\qquad\qquad\qquad \forall w.\, F(x,w) \wedge F(y,w) \Rightarrow w = z) \\ & \Rightarrow \exists u.\, \forall v.\, \neg(v = u) \Rightarrow F(u,v)) \end{array}$$

The theory of equality

Even equational logic is undecidable, but the purely universal (quantiifer-free) fragment is decidable. For example:
$\forall x.\, f(f(f(x)) = x \land f(f(f(f(f(x))))) = x \Rightarrow f(x) = x$
after negating and Skolemizing we need to test a ground formula for satisfiability:
$f(f(f(c)) = c \land f(f(f(f(f(c))))) = c \land \neg(f(c) = c)$
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.

Current first-order provers

There are numerous competing first-order theorem provers

- Vampire
- E
- SPASS
- Prover9
- LeanCop

and many specialist equational solvers like Waldmeister and EQP. There are annual theorem-proving competitions where they are tested against each other, which has helped to drive progress.

3: Decidable problems in logic and algebra

John Harrison, Intel Corporation
Computer Science Club, St. Petersburg
Sun 29th September 2013 (11:15–12:50)

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:

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

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)
- Complex arithmetic: arithmetic equations with addition and multiplication interpreted over $\mathbb{C}$ (or other algebraically closed field of characteristic 0).

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)
- Complex arithmetic: arithmetic equations with addition and multiplication interpreted over $\mathbb{C}$ (or other algebraically closed field of characteristic 0).

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.

Word problems

Want to decide whether one set of equations implies another in a class of algebraic structures:

$$\forall \overline{x}.\ s_1 = t_1 \wedge \cdots \wedge s_n = t_n \Rightarrow s = t$$

For rings, we can assume it's a standard polynomial form

$$\forall \overline{x}.\ p_1(\overline{x}) = 0 \wedge \cdots \wedge p_n(\overline{x}) = 0 \Rightarrow q(\overline{x}) = 0$$

Word problem for rings

$$\forall \overline{x}.\ p_1(\overline{x}) = 0 \wedge \cdots \wedge p_n(\overline{x}) = 0 \Rightarrow q(\overline{x}) = 0$$

holds in all rings iff

$$q \in \mathsf{Id}_{\mathbb{Z}} \langle p_1, \ldots, p_n \rangle$$

i.e. there exist 'cofactor' polynomials with integer coefficients such that

$$p_1 \cdot q_1 + \cdots + p_n \cdot q_n = q$$

Special classes of rings

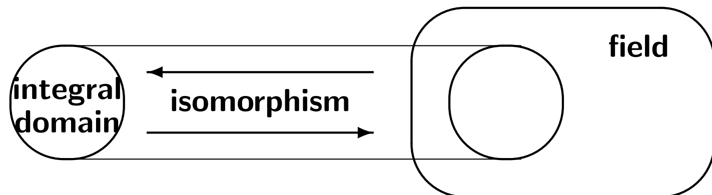
- Torsion-free: $\overbrace{x + \cdots + x}^{n \text{ times}} = 0 \Rightarrow x = 0$ for $n \geq 1$
- Characteristic p: $\overbrace{1 + \cdots + 1}^{n \text{ times}} = 0$ iff $p | n$
- Integral domains: $x \cdot y = 0 \Rightarrow x = 0 \vee y = 0$ (and $1 \neq 0$).

Special word problems

$$\forall \overline{x}.\ p_1(\overline{x}) = 0 \wedge \cdots \wedge p_n(\overline{x}) = 0 \Rightarrow q(\overline{x}) = 0$$

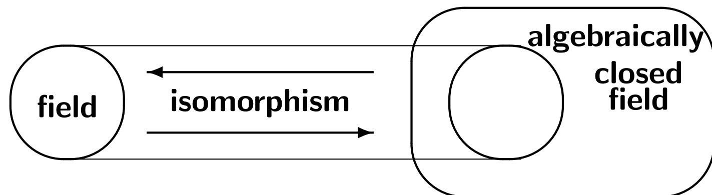
- Holds in all rings iff $q \in \mathsf{Id}_{\mathbb{Z}} \langle p_1, \ldots, p_n \rangle$
- Holds in all torsion-free rings iff $q \in \mathsf{Id}_{\mathbb{Q}} \langle p_1, \ldots, p_n \rangle$
- Holds in all integral domains iff $q^k \in \mathsf{Id}_{\mathbb{Z}} \langle p_1, \ldots, p_n \rangle$ for some $k \geq 0$
- Holds in all integral domains of characteristic 0 iff $q^k \in \mathsf{Id}_{\mathbb{Q}} \langle p_1, \ldots, p_n \rangle$ for some $k \geq 0$

Embedding in field of fractions



Universal formula in the language of rings holds in all integral domains [of characteristic p] iff it holds in all fields [of characteristic p].

Embedding in algebraic closure



Universal formula in the language of rings holds in all fields [of characteristic p] iff it holds in all algebraically closed fields [of characteristic p]

Connection to the Nullstellensatz

Also, algebraically closed fields of the same characteristic are elementarily equivalent.
For a universal formula in the language of rings, all these are equivalent:

- It holds in all integral domains of characteristic 0
- It holds in all fields of characteristic 0
- It holds in all algebraically closed fields of characteristic 0
- It holds in any given algebraically closed field of characteristic 0
- It holds in $\mathbb{C}$

Penultimate case is basically the Hilbert Nullstellensatz.

Gröbner bases

Can solve all these ideal membership goals in various ways.
The most straightforward uses *Gröbner bases*.
Use polynomial $m_1 + m_2 + \cdots + m_p = 0$ as a rewrite rule $m_1 = -m_2 + \cdots + -m_p$ for a 'head' monomial according to ordering.
Perform operation analogous to Knuth-Bendix completion to get expanded set of equations that is confluent, a *Gröbner basis*.

Geometric theorem proving

In principle can solve most geometric problems by using coordinate translation then Tarski's real quantifier elimination.

Example: A, B, C are collinear iff

$(A_x - B_x)(B_y - C_y) = (A_y - B_y)(B_x - C_x)$

In practice, it's much faster to use decision procedures for complex numbers. Remarkably, many geometric theorems remain true in this more general context.

As well as Gröbner bases, Wu pioneered the approach using characteristic sets (Ritt-Wu triangulation).

Degenerate cases

Many simple and not-so-simple theorems can be proved using a straightforward algebraic reduction, but we may encounter problems with *degenerate cases*, e.g.

> *The parallelegram theorem: If $ABCD$ is a parallelogram, and the diagonals AC and BD meet at E, then $|AE| = |CE|$.*

Degenerate cases

Many simple and not-so-simple theorems can be proved using a straightforward algebraic reduction, but we may encounter problems with *degenerate cases*, e.g.

> *The parallelegram theorem: If $ABCD$ is a parallelogram, and the diagonals AC and BD meet at E, then $|AE| = |CE|$.*

This is 'true' but fails when ABC is collinear.

Degenerate cases

Many simple and not-so-simple theorems can be proved using a straightforward algebraic reduction, but we may encounter problems with *degenerate cases*, e.g.

> *The parallelegram theorem: If ABCD is a parallelogram, and the diagonals AC and BD meet at E, then* $|AE| = |CE|$.

This is 'true' but fails when ABC is collinear.
A major strength of Wu's method is that it can actually derive many such conditions automatically without the user's having to think of them.

Quantifier elimination for real-closed fields

Take a first-order language:

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

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 $p_1(x)$, ..., $p_n(x)$.
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: $\forall x.\, x \geq 0 \Rightarrow \exists y.\, x = y^2$
- 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.

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:

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

and this one in Presburger arithmetic defines squaring:

$(\forall n.\, f(-n) = f(n)) \wedge (f(0) = 0) \wedge$

$(\forall n.\, 0 \leq n \Rightarrow f(n+1) = f(n) + n + n + 1)$

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:

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

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

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

so now we can split the conjuncts according to signature:

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

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, no disjunctions.
- Component procedures can actually produce equational consequences rather than waiting passively for formulas to test.

Shostak's method

Can be seen as an optimization of Nelson-Oppen method for common special cases. Instead of just a decision method each component theory has a

- Canonizer — puts a term in a T-canonical form
- Solver — solves systems of equations

Shostak's original procedure worked well, but the theory was flawed on many levels. In general his procedure was incomplete and potentially nonterminating.
It's only recently that a full understanding has (apparently) been reached.

SMT

Recently the trend has been to use a SAT solver as the core of combined decision procedures (SMT = "satisfiability modulo theories").

- Use SAT to generate propositionally satisfiable assignments
- Use underlying theory solvers to test their satisfiability in the theories.
- Feed back conflict clauses to SAT solver

Mostly justified using same theory as Nelson-Oppen, and likewise a modular structure where new theories can be added

- Arrays
- Machine words
- ...

4: Interactive theorem proving and proof checking

John Harrison, Intel Corporation
Computer Science Club, St. Petersburg
Sun 29th September 2013 (13:00–14:35)

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)

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

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

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

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

$$\vdash \quad p \Rightarrow \forall x.\ p \quad [\textbf{Provided } x \notin \mathrm{FV}(p)]$$

$$\vdash \quad (\exists x.\ x = t) \quad [\textbf{Provided } x \notin \mathrm{FVT}(t)]$$

$$\vdash \quad t = t$$

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

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

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 (Tobias's lectures)

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.

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

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

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?

5: Applications to mathematics and computer verification

John Harrison, Intel Corporation
Computer Science Club, St. Petersburg
Sun 29th September 2013 (15:35–17:10)

100 years since Principia Mathematica

Principia Mathematica was the first sustained and successful actual formalization of mathematics.

100 years since Principia Mathematica

Principia Mathematica was the first sustained and successful actual formalization of mathematics.

- This practical formal mathematics was to forestall objections to Russell and Whitehead's 'logicist' thesis, not a goal in itself.

100 years since Principia Mathematica

Principia Mathematica was the first sustained and successful actual formalization of mathematics.

- This practical formal mathematics was to forestall objections to Russell and Whitehead's 'logicist' thesis, not a goal in itself.
- The development was difficult and painstaking, and has probably been studied in detail by very few.

100 years since Principia Mathematica

Principia Mathematica was the first sustained and successful actual formalization of mathematics.

- This practical formal mathematics was to forestall objections to Russell and Whitehead's 'logicist' thesis, not a goal in itself.
- The development was difficult and painstaking, and has probably been studied in detail by very few.
- Subsequently, the idea of actually formalizing proofs has not been taken very seriously, and few mathematicians do it today.

100 years since Principia Mathematica

Principia Mathematica was the first sustained and successful actual formalization of mathematics.

- This practical formal mathematics was to forestall objections to Russell and Whitehead's 'logicist' thesis, not a goal in itself.
- The development was difficult and painstaking, and has probably been studied in detail by very few.
- Subsequently, the idea of actually formalizing proofs has not been taken very seriously, and few mathematicians do it today.

But thanks to the rise of the computer, the actual formalization of mathematics is attracting more interest.

The importance of computers for formal proof

Computers can both help *with* formal proof and give us new reasons to be interested in it:

The importance of computers for formal proof

Computers can both help *with* formal proof and give us new reasons to be interested in it:

- Computers are expressly designed for performing formal manipulations quickly and without error, so can be used to check and partly generate formal proofs.

The importance of computers for formal proof

Computers can both help *with* formal proof and give us new reasons to be interested in it:

- Computers are expressly designed for performing formal manipulations quickly and without error, so can be used to check and partly generate formal proofs.
- Correctness questions in computer science (hardware, programs, protocols etc.) generate a whole new array of difficult mathematical and logical problems where formal proof can help.

The importance of computers for formal proof

Computers can both help *with* formal proof and give us new reasons to be interested in it:

- Computers are expressly designed for performing formal manipulations quickly and without error, so can be used to check and partly generate formal proofs.
- Correctness questions in computer science (hardware, programs, protocols etc.) generate a whole new array of difficult mathematical and logical problems where formal proof can help.

Because of these dual connections, interest in formal proofs is strongest among computer scientists, but some 'mainstream' mathematicians are becoming interested too.

Russell was an early fan of mechanized formal proof

Newell, Shaw and Simon in the 1950s developed a 'Logic Theory Machine' program that could prove some of the theorems from *Principia Mathematica* automatically.

Russell was an early fan of mechanized formal proof

Newell, Shaw and Simon in the 1950s developed a 'Logic Theory Machine' program that could prove some of the theorems from *Principia Mathematica* automatically.

> *"I am delighted to know that Principia Mathematica can now be done by machinery [...] I am quite willing to believe that everything in deductive logic can be done by machinery. [...] I wish Whitehead and I had known of this possibility before we wasted 10 years doing it by hand." [letter from Russell to Simon]*

Russell was an early fan of mechanized formal proof

Newell, Shaw and Simon in the 1950s developed a 'Logic Theory Machine' program that could prove some of the theorems from *Principia Mathematica* automatically.

> *"I am delighted to know that Principia Mathematica can now be done by machinery [...] I am quite willing to believe that everything in deductive logic can be done by machinery. [...] I wish Whitehead and I had known of this possibility before we wasted 10 years doing it by hand." [letter from Russell to Simon]*

Newell and Simon's paper on a more elegant proof of one result in PM was rejected by JSL because it was co-authored by a machine.

Formalization in current mathematics

Traditionally, we understand *formalization* to have two components, corresponding to Leibniz's *characteristica universalis* and *calculus ratiocinator*.

Formalization in current mathematics

Traditionally, we understand *formalization* to have two components, corresponding to Leibniz's *characteristica universalis* and *calculus ratiocinator*.

- Express *statements* of theorems in a formal language, typically in terms of primitive notions such as sets.

Formalization in current mathematics

Traditionally, we understand *formalization* to have two components, corresponding to Leibniz's *characteristica universalis* and *calculus ratiocinator*.

- Express *statements* of theorems in a formal language, typically in terms of primitive notions such as sets.
- Write *proofs* using a fixed set of formal inference rules, whose correct form can be checked algorithmically.

Formalization in current mathematics

Traditionally, we understand *formalization* to have two components, corresponding to Leibniz's *characteristica universalis* and *calculus ratiocinator*.

- Express *statements* of theorems in a formal language, typically in terms of primitive notions such as sets.
- Write *proofs* using a fixed set of formal inference rules, whose correct form can be checked algorithmically.

Correctness of a formal proof is an objective question, algorithmically checkable in principle.

Mathematics is reduced to sets

The explication of mathematical concepts in terms of sets is now quite widely accepted (see *Bourbaki*).

- A real number is a set of rational numbers ...
- A Turing machine is a quintuple $(\Sigma, A, \ldots)$

Statements in such terms are generally considered clearer and more objective. (Consider pathological functions from real analysis ...)

Symbolism is important

The use of symbolism in mathematics has been steadily increasing over the centuries:

> *"[Symbols] have invariably been introduced to make things easy. [. . .] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [. . .] Civilisation advances by extending the number of important operations which can be performed without thinking about them." (Whitehead,* An Introduction to Mathematics*)*

Formalization is the key to rigour

Formalization now has a important conceptual role in principle:

> *". . . the correctness of a mathematical text is verified by comparing it, more or less explicitly, with the rules of a formalized language." (Bourbaki,* Theory of Sets*)*
>
> *"A Mathematical proof is rigorous when it is (or could be) written out in the first-order predicate language $L(\in)$ as a sequence of inferences from the axioms ZFC, each inference made according to one of the stated rules." (Mac Lane,* Mathematics: Form and Function*)*

What about in practice?

Mathematicians don't use logical symbols

Variables were used in logic long before they appeared in mathematics, but logical symbolism is rare in current mathematics. Logical relationships are usually expressed in natural language, with all its subtlety and ambiguity.
Logical symbols like '$\Rightarrow$' and '$\forall$' are used *ad hoc*, mainly for their abbreviatory effect.

> *"as far as the mathematical community is concerned George Boole has lived in vain" (Dijkstra)*

Mathematicians don't do formal proofs ...

The idea of actual formalization of mathematical proofs has not been taken very seriously:

> *"this mechanical method of deducing some mathematical theorems has no practical value because it is too complicated in practice." (Rasiowa and Sikorski,* The Mathematics of Metamathematics*)*
>
> *"[...] the tiniest proof at the beginning of the Theory of Sets would already require several hundreds of signs for its complete formalization. [...] formalized mathematics cannot in practice be written down in full [...] We shall therefore very quickly abandon formalized mathematics" (Bourbaki,* Theory of Sets*)*

…and the few people that do end up regretting it

> *"my intellect never quite recovered from the strain of writing [*Principia Mathematica*]. I have been ever since definitely less capable of dealing with difficult abstractions than I was before." (Russell,* Autobiography*)*

However, now we have computers to check and even automatically generate formal proofs.

Our goal is now not so much philosphical, but to achieve a real, practical, useful increase in the precision and accuracy of mathematical proofs.

Are proofs in doubt?

Mathematical proofs are subjected to peer review, but errors often escape unnoticed.

> *"Professor Offord and I recently committed ourselves to an odd mistake (Annals of Mathematics (2) 49, 923, 1.5). In formulating a proof a plus sign got omitted, becoming in effect a multiplication sign. The resulting false formula got accepted as a basis for the ensuing fallacious argument. (In defence, the final result was known to be true.)" (Littlewood,* Miscellany*)*

A book by Lecat gave 130 pages of errors made by major mathematicians up to 1900.
A similar book today would no doubt fill many volumes.

Even elegant textbook proofs can be wrong

"The second edition gives us the opportunity to present this new version of our book: It contains three additional chapters, substantial revisions and new proofs in several others, as well as minor amendments and improvements, many of them based on the suggestions we received. It also misses one of the old chapters, about the "problem of the thirteen spheres," whose proof turned out to need details that we couldn't complete in a way that would make it brief and elegant." (Aigner and Ziegler, Proofs from the Book*)*

Most doubtful informal proofs

What are the proofs where we do in practice worry about correctness?

- Those that are just very long and involved. Classification of finite simple groups, Seymour-Robertson graph minor theorem
- Those that involve extensive computer checking that cannot in practice be verified by hand. Four-colour theorem, Hales's proof of the Kepler conjecture
- Those that are about very technical areas where complete rigour is painful. Some branches of proof theory, formal verification of hardware or software

4-colour Theorem

Early history indicates fallibility of the traditional social process:

- Proof claimed by Kempe in 1879
- Flaw only point out in print by Heaywood in 1890

Later proof by Appel and Haken was apparently correct, but gave rise to a new worry:

- How to assess the correctness of a proof where many explicit configurations are checked by a computer program?

Most worries finally dispelled by Gonthier's formal proof in Coq.

Recent formal proofs in pure mathematics

Three notable recent formal proofs in pure mathematics:

- Prime Number Theorem — Jeremy Avigad et al (Isabelle/HOL), John Harrison (HOL Light)
- Jordan Curve Theorem — Tom Hales (HOL Light), Andrzej Trybulec et al. (Mizar)
- Four-colour theorem — Georges Gonthier (Coq)
- Odd order theorem — Georges Gonthier and others (Coq)

These indicate that highly non-trivial results are within reach. However these all required months/years of work.

The Kepler conjecture

The *Kepler conjecture* states that no arrangement of identical balls in ordinary 3-dimensional space has a higher packing density than the obvious ‘cannonball’ arrangement.
Hales, working with Ferguson, arrived at a proof in 1998:

- 300 pages of mathematics: geometry, measure, graph theory and related combinatorics, ...
- 40,000 lines of supporting computer code: graph enumeration, nonlinear optimization and linear programming.

Hales submitted his proof to *Annals of Mathematics* ...

The response of the reviewers

After a full four years of deliberation, the reviewers returned:

> *"The news from the referees is bad, from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for.*
> *Fejes Toth thinks that this situation will occur more and more often in mathematics. He says it is similar to the situation in experimental science — other scientists acting as referees can't certify the correctness of an experiment, they can only subject the paper to consistency checks. He thinks that the mathematical community will have to get used to this state of affairs."*

The birth of Flyspeck

Hales's proof was eventually published, and no significant error has been found in it. Nevertheless, the verdict is disappointingly lacking in clarity and finality.

As a result of this experience, the journal changed its editorial policy on computer proof so that it will no longer even try to check the correctness of computer code.

Dissatisfied with this state of affairs, Hales initiated a project called *Flyspeck* to completely formalize the proof.

Flyspeck

Flyspeck = 'Formal Proof of the Kepler Conjecture'.

> *"In truth, my motivations for the project are far more complex than a simple hope of removing residual doubt from the minds of few referees. Indeed, I see formal methods as fundamental to the long-term growth of mathematics. (Hales,* The Kepler Conjecture*)*

The formalization effort has been running for a few years now with a significant group of people involved, some doing their PhD on Flyspeck-related formalization.
In parallel, Hales has simplified the informal proof using ideas from Marchal, significantly cutting down on the formalization work.

Flyspeck: current status

- Almost all the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings.
- Many of the linear programs have been verified in Isabelle/HOL by Steven Obua. Alexey Solovyev has recently developed a faster HOL Light formalization.
- The graph enumeration process has been verified (and improved in the process) by Tobias Nipkow in Isabelle/HOL
- Some initial work by Roland Zumkeller on nonlinear part using Bernstein polynomials. Solovyev has been working on formalizing this in HOL Light.

Formal verification

In most software and hardware development, we lack even *informal* proofs of correctness.

Correctness of hardware, software, protocols etc. is routinely "established" by testing.

However, exhaustive testing is impossible and subtle bugs often escape detection until it's too late.

The consequences of bugs in the wild can be serious, even deadly.

Formal verification (*proving* correctness) seems the most satisfactory solution, but gives rise to large, ugly proofs.

Recent formal proofs in computer system verification

Some successes for verification using theorem proving technology:

- Microcode algorithms for floating-point division, square root and several transcendental functions on Intel® Itanium® processor family (John Harrison, HOL Light)
- CompCert verified compiler from significant subset of the C programming language into PowerPC assembler (Xavier Leroy et al., Coq)
- Designed-for-verification version of L4 operating system microkernel (Gerwin Klein et al., Isabelle/HOL).

Again, these indicate that complex and subtle computer systems can be verified, but significant manual effort was needed, perhaps tens of person-years for L4.

A diversity of activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

A diversity of activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

If the Intel® Software and Services Group (SSG) were split off as a separate company, it would be in the top 10 software companies worldwide.

A diversity of verification problems

This gives rise to a corresponding diversity of verification problems, and of verification solutions.

- Propositional tautology/equivalence checking (FEV)
- Symbolic simulation
- Symbolic trajectory evaluation (STE)
- Temporal logic model checking
- Combined decision procedures (SMT)
- First order automated theorem proving
- Interactive theorem proving

Most of these techniques (trading automation for generality / efficiency) are in active use at Intel.

A spectrum of formal techniques

Traditionally, formal verification has been focused on complete proofs of functional correctness.

But recently there have been notable successes elsewhere for 'semi-formal' methods involving abstraction or more limited property checking.

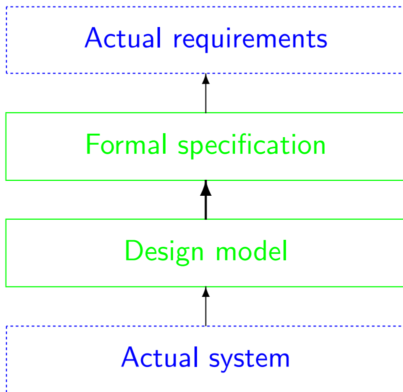
- Airbus A380 avionics
- Microsoft SLAM/SDV

One can also consider applying theorem proving technology to support testing or other traditional validation methods like path coverage.

These are all areas of interest at Intel.

Models and their validation

We have the usual concerns about validating our specs, but also need to pay attention to the correspondence between our models and physical reality.



Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

- In 1978, Intel encountered problems with ‘soft errors’ in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

However, these are rare and apparently well controlled by existing engineering best practice.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $475 million to cover the costs.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $475 million to cover the costs.

This did at least considerably improve investment in formal verification.

Layers of verification

If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

Layers of verification

If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

- Implement high-level floating-point algorithm assuming addition works correctly.

Layers of verification

If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

- Implement high-level floating-point algorithm assuming addition works correctly.
- Implement a cache coherence protocol assuming that the abstract protocol ensures coherence.

Layers of verification

If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

- Implement high-level floating-point algorithm assuming addition works correctly.
- Implement a cache coherence protocol assuming that the abstract protocol ensures coherence.

Many similar ideas all over computing: protocol stack, virtual machines etc.

Layers of verification

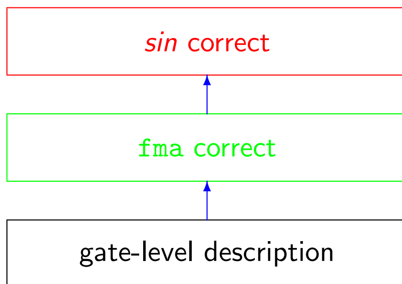
If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

- Implement high-level floating-point algorithm assuming addition works correctly.
- Implement a cache coherence protocol assuming that the abstract protocol ensures coherence.

Many similar ideas all over computing: protocol stack, virtual machines etc.
If this clean separation starts to break down, we may face much worse verification problems...

How some of our verifications fit together

For example, the `fma` behavior is the *assumption* for my verification, and the *conclusion* for someone else's.



But this is not quite trivial when the verifications use different formalisms!

Our work

We have formally verified correctness of various floating-point algorithms.

- Division and square root (Marstein-style, using fused multiply-add to do Newton-Raphson or power series approximation with delicate final rounding).
- Transcendental functions like *log* and *sin* (table-driven algorithms using range reduction and a core polynomial approximations).

Proofs use the HOL Light prover

- `http://www.cl.cam.ac.uk/users/jrh/hol-light`

Our HOL Light proofs

The mathematics we formalize is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

Needs several special-purpose proof procedures, e.g.

- Verifying solution set of some quadratic congruences
- Proving primality of particular numbers
- Proving bounds on rational approximations
- Verifying errors in polynomial approximations

Example: tangent algorithm

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.

Example: tangent algorithm

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.
- If the reduced argument r is still not small enough, it is separated into its leading few bits B and the trailing part $x = r - B$, and the overall result computed from $tan(x)$ and pre-stored functions of B, e.g.

$$tan(B + x) = tan(B) + \frac{\frac{1}{sin(B)cos(B)} tan(x)}{cot(B) - tan(x)}$$

Example: tangent algorithm

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.
- If the reduced argument r is still not small enough, it is separated into its leading few bits B and the trailing part $x = r - B$, and the overall result computed from $tan(x)$ and pre-stored functions of B, e.g.

$$tan(B + x) = tan(B) + \frac{\frac{1}{sin(B)cos(B)} tan(x)}{cot(B) - tan(x)}$$

- Now a power series approximation is used for $tan(r)$, $cot(r)$ or $tan(x)$ as appropriate.

Overview of the verification

To verify this algorithm, we need to prove:

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.
- The rounding errors involved in computing with floating point arithmetic are within bounds.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.
- The rounding errors involved in computing with floating point arithmetic are within bounds.

Most of these parts are non-trivial. Moreover, some of them require more pure mathematics than might be expected.

Why mathematics?

Controlling the error in range reduction becomes difficult when the reduced argument $X - N\pi/2$ is small.
To check that the computation is accurate enough, we need to know:

> *How close can a floating point number be to an integer multiple of $\pi/2$?*

Even deriving the power series (for $0 < |x| < \pi$):

$$cot(x) = 1/x - \frac{1}{3}x - \frac{1}{45}x^3 - \frac{2}{945}x^5 - \ldots$$

is much harder than you might expect.

Why HOL Light?

We need a general theorem proving system with:

- High standard of logical rigor and reliability
- Ability to mix interactive and automated proof
- Programmability for domain-specific proof tasks
- A substantial library of pre-proved mathematics

Other theorem provers such as ACL2, Coq and PVS have also been used for verification in this area.