Automated Theorem Proving: a Technology Roadmap

Lawrence C Paulson FRS
1. Proof Assistants
Mechanising a formal logic

- **Syntax**: a precise specification of the formalism’s grammar
- **Semantics**: the mathematical meaning of logical terms and formulas
- **Proof theory**: a precise calculus for deriving or verifying true formulas
- **Automation**: algorithms and data structures to verify formulas efficiently
A variety of verification technologies

- SAT solving (originated in the 1960s, revived in the 1990s) for Boolean logic
- SMT solving: extending SAT with arithmetic, arrays, quantifiers and more
- BDDs: a powerful data structure for large Boolean problems
- Resolution, for first-order logic (quantifiers): logical reasoning + rewriting

each of these can handle large problems and is fully automatic
So why *interactive* theorem proving?

- No automatic method can prove even quite simple statements
  - *there are infinitely many prime numbers;* $\sqrt{2}$ *is irrational*
- Only higher-order formalisms are expressive enough
- Real-world projects require large hierarchies of specifications
  - “interactive theorem provers” should be called *specification editors*
Why do interactive provers need automation?

- Even the simplest facts are extremely tedious to prove in a basic calculus
- Lengthy calculations drawing on thousands of facts
- Almost unlimited computer power could reduce the burden on users
  - finding new proofs (by classical theorem proving)
  - identifying similar proofs in existing libraries (by machine learning)
Interactive theorem provers today

- **Simple types (higher-order logic):** Isabelle/HOL, HOL4, HOL Light
  - a simpler but weaker formal calculus
  - straightforward automation
  - can express sophisticated constructions

- **Dependent types:** Lean, Coq, Agda
  - formally stronger and more expressive calculi
  - constructive proof
  - popular with mathematicians and theoreticians
The LCF Architecture

- A small kernel implements the logic and has the sole power to generate theorems (Milner, 1979)
- ... safety ensured by the programming language’s abstract data types.
- All specification methods and proof procedures expand to full proofs.
- Unsoundness is less likely, but the implementation is more complicated.
- Adopted by HOL, Isabelle, Coq, Lean… but not PVS, ACL2
Common features in *all* proof assistants

- A *language* for declaring types & definitions, stating theorems
- *Recursive* functions and types
- A system of *proof tactics*
- A dependency graph for theories

- A modern user interface supporting *subgoal-oriented* proof
- *Automation*: rewriting, arithmetic and specialist proof procedures
- Code extraction / generation
- Extensive libraries of basic maths
2. Isabelle/HOL
Some distinctive features of Isabelle/HOL

- Classical proof search using forward/backward chaining
- Quickcheck and nitpick: powerful counterexample detection
- Sledgehammer: a link to external provers
- Isar, a readable language for structured proofs
- Extensive exploitation of parallelism
Higher-order logic

- First-order logic extended with **polymorphic types, functions and sets**
- A type of *truth values*, with no distinction between terms and formulas
- Expressive enough to formalise sophisticated mathematical definitions
- Easy to understand and implement

“HOL = functional programming + logic”
Classical proof search (auto, force, blast …)

- forward or backward chaining using hundreds of built-in facts about logic, sets, simple maths and data structures
- easily augmented by the user to support their own development
- both automatic and interactive modes

\[(\exists y \forall x. P_{xy} \leftrightarrow P_{xx}) \rightarrow \neg \forall x \exists y \forall z. P_{zy} \leftrightarrow \neg P_{zx}\]

\[\left( \bigcup_{i \in I} A_i \cup B_i \right) = \left( \bigcup_{i \in I} A_i \right) \cup \left( \bigcup_{i \in I} B_i \right)\]

This was the key to all the work verifying cryptographic protocols.
Quickcheck and nitpick

Because many theorems are stated incorrectly

- **Quickcheck** detects false statements by evaluation with appropriate test data and also by symbolic evaluation [it excels at inductive datatypes]
- Nitpick detects false statements using sophisticated translations into first-order relational logic, using the SAT-based Kodkod model finder
- inductive/coinductive predicates and other advanced constructions are permitted
Sledgehammer

- Calls several external provers to work on the current goal
- ... but does not trust their proofs!
- Zero configuration and 1-click invocation
- Access to the whole lemma library, able to dig up the most obscure facts
- Particularly powerful in conjunction with *structured proofs*
3. Structured Proofs
Tactic proofs: fit only for machines

**Mean value theorem**

```
let MVT_LEMMA = prove(  
  \(\forall f:real\to real\ a\ b. \  
  (\x. f(x) - (((f(b) - f(a)) / (b - a)) \times x)) (a) = (\x. f(x) - (((f(b) - f(a)) / (b - a)) \times x)) (b), \)
  REPEAT GEN_TAC THEN BETA_TAC THEN  
  ASM_CASES_TAC `b:real = a` THEN ASM_REWRITE_TAC[] THEN  
  ONCE_REWRITE_TAC[REAL_MUL_SYM] THEN  
  RULE_ASSUM_TAC(ONCE_REWRITE_RULE[GSYM REAL_SUB_0]) THEN  
  MP_TAC(GENL [`x:real`; `y:real`](SPECL [`x:real`; `y:real`; `b - a`]
  REAL_EQ_RMUL)) THEN  
  ASM_REWRITE_TAC[] THEN  
  DISCH_THEN(fun th -> GEN_REWRITE_TAC I [GSYM th]) THEN  
  REWRITE_TAC[REAL_SUB_RDISTRIB; GSYM REAL_MUL_ASSOC] THEN  
  FIRST_ASSUM(fun th -> REWRITE_TAC[MATCH_MP REAL_DIV_RMUL th]) THEN  
  GEN_REWRITE_TAC(RAND_CONV to RAND_CONV) [REAL_MUL_SYM] THEN  
  GEN_REWRITE_TAC(LAND_CONV to RAND_CONV) [REAL_MUL_SYM] THEN  
  REWRITE_TAC[REAL_SUB; REAL_LDISTRIB; REAL_RDISTRIB] THEN  
  REWRITE_TAC[GSYM REAL_NEG_LMUL; GSYM REAL_NEG_RMUL; REAL_NEG_ADD; REAL_NEG_NEG] THEN  
  REWRITE_TAC[GSYM REAL_ADD_ASSOC] THEN  
  ALL_TAC);  
```

```
let MVT = prove(  
  `\(\forall f:real\to real\ a\ b. \  
  (\x. f(x) - (((f(b) - f(a)) / (b - a)) \times x)) (a) = (\x. f(x) - (((f(b) - f(a)) / (b - a)) \times x)) (b)\)  
  ==> \[l z. a < z \land z < b \land (f'(z)) = ((f(b) - f(a)) / (b - a)) \times l\]
  ONCE_REWRITE_TAC[REAL_MUL_SYM] THEN  
  REPEAT GEN_TAC THEN STRIP_TAC THEN  
  MP_TAC(SPECL [`(\x. f(x) - (((f(b) - f(a)) / (b - a)) \times x)`; `a:real`; `b:real`]
  REAL_EQ_RMUL) THEN  
  W(CONJ_TAC SUBGOAL_THEN (fun t -> REWRITE_TAC[t] o funpow 2 (fst o dest_imp) o snd) THENL  
  [ASM_REWRITE_TAC[MVT_LEMMA] THEN BETA_TAC THEN  
  CONJ_TAC THEN X_GEN_TAC `x:real` THENL  
  [DISCH_TAC THEN CONV_TAC(ONCE_DEPTH_CONV HABS_CONV) THEN  
  MATCH_MP_TAC CONT_SUB THEN CONJ_TAC THENL  
  [CONV_TAC(ONCE_DEPTH_CONV ETA_CONV) THEN  
  FIRST_ASSUM MATCH_MP_TAC THEN ASM_REWRITE_TAC[];
  CONV_TAC(ONCE_DEPTH_CONV HABS_CONV) THEN MATCH_MP_TAC CONT_MUL THEN  
  REWRITE_TAC[CONT_CONST] THEN MATCH_MP_TAC DIFF_CONT THEN  
  EXISTS_TAC `&1` THEN MATCH_ACCEPT_TAC DIFF_X];
  DISCH_THEN(fun th -> FIRST_ASSUM(MP_TAC to C MATCH_MP th)) THEN  
  ASM_REWRITE_TAC[] THEN  
  DISCH_THEN,X_CHOOSE_TAC `l:real` THEN  
  EXISTS_TAC `(l - ((f(b) - f(a)) / (b - a)))` THEN  
  CONV_TAC(ONCE_DEPTH_CONV HABS_CONV) THEN MATCH_MP_TAC DIFF_CMUL THEN MATCH_ACCEPT_TAC DIFF_X];
  ALL_TAC];  
```
The same, as a \textit{structured} proof

\begin{verbatim}
theorem mvt:
  fixes φ :: "real ⇒ real"
  assumes "a < b"
  and contf: "continuous_on {a..b} φ"
  and derf: "∀x. [a < x; x < b] ⇒ (φ has_derivative φ' x) (at x)"
  obtains ξ where "a < ξ" "ξ < b" "φ b - φ a = (φ' ξ) (b-a)"
proof -
  define f where "f ≡ λx. φ x - (φ b - φ a) / (b-a) * x"
  have "∃ξ. a < ξ ∧ ξ < b ∧ (λy. φ' ξ y - (φ b - φ a) / (b-a) * y) = (λv. 0)"" proof (intro Rolle_deriv[OF ⟨a < b⟩])
    fix x
    assume x: "a < x" "x < b"
    show "(f has_derivative (λy. φ' x y - (φ b - φ a) / (b-a) * y)) (at x)" unfolding f_def by (intro derivative_intros derf x)
  next
    show "f a = f b"
    using assms by (simp add: f_def field_simps)
  next
    show "continuous_on {a..b} f"
    unfolding f_def by (intro continuous_intros assms)
  qed
then show ?thesis
  by (smt (verit, ccfv_SIG) pos_le_divide_eq pos_less_divide_eq that)
qed
\end{verbatim}
Structured proofs are necessary!

- Because formal proofs should *make sense to users*

  ... reducing the need to **trust** our verification tools

- For *reuse* and eventual *translation* to other systems

- For *maintenance* (easily fix proofs that break due to changes to definitions… or **automation**)

*With some other systems,*

**users avoid automation for that reason!**
Structured proofs assist machine learning!

- Working **locally** within a large proof
- Looking for just the **next step** (not the whole proof)
- Proof by analogy
- Identifying **idioms**
For Isabelle, we’ve lots of data

- About 230K proof lines in Isabelle’s maths libraries: Analysis, Complex Analysis, Number Theory, Algebra
- Nearly 3.4M proof lines nearly 700 entries in the Archive of Formal Proofs (not all mathematics though)
- Over 400 different authors: diverse styles and topics
Lots of structured “chunks”

- Structured proof fragments contain *explicit assertions* and *context elements* that could drive learning

- These might relate to *natural mathematical steps*

- Proving a function to be continuous

- Getting a ball around a point within an open set

- Covering a compact set with finitely many balls
It is essential to *synthesise terms and formulas*

Even tactics take arguments

Structured proofs mostly consist of explicit formulas
4. A Few Proof Idioms for ML
Inequality chains

```
have "\|X \cdot m * Y \cdot m - X \cdot n * Y \cdot n\| = \|X \cdot m * (Y \cdot m - Y \cdot n) + (X \cdot m - X \cdot n) * Y \cdot n\|",
    unfolding mult_diff_mult ..
also have "... ≤ \|X \cdot m * (Y \cdot m - Y \cdot n)\| + \|(X \cdot m - X \cdot n) * Y \cdot n\|",
    by (rule abs_triangle_ineq)
also have "... = \|X \cdot m\| * \|Y \cdot m - Y \cdot n\| + \|X \cdot m - X \cdot n\| * \|Y \cdot n\|",
    unfolding abs_mult ..
also have "... < a * t + s * b",
    by (simp_all add: add_strict_mono mult_strict_mono' a b i j *)
finally show "\|X \cdot m * Y \cdot m - X \cdot n * Y \cdot n\| < r",
    by (simp only: r)
```

typically by the \textit{triangle inequality}

with simple algebraic manipulations

there are hundreds of examples
Simple topological steps

have "open (interior I)" by auto
from openE[OF this ⟨x ∈ interior I⟩]
obtain e where e: "0 < e" "ball x e ⊆ interior I".

define U where "U = (λw. (w - ξ) * g w) \ T"
have "open U" by (metis oimT U_def)
moreover have "0 ∈ U"
  using ⟨ξ ∈ T⟩ by (auto simp: U_def intro: image_eqI [where x = ξ])
ultimately obtain ε where "ε>0" and ε: "cball 0 ε ⊆ U"
  using ⟨open U⟩ open_contains_cball by blast

a neighbourhood around a point within an open set

many similar but not identical instances
have "real (Suc n) *\text{R} S (x + y) (Suc n) = (x + y) * (\sum_{i \leq n.} S \times i * S y (n - i))" by (metis Suc.hyps times_S)
also have "... = x * (\sum_{i \leq n.} S \times i * S y (n - i)) + y * (\sum_{i \leq n.} S \times i * S y (n - i))" by (rule distrib_right)
also have "... = (\sum_{i \leq n.} x \times S \times i * S y (n - i)) + (\sum_{i \leq n.} S \times i * y * S y (n - i))" by (simp add: sum_distrib_left ac_simps S_comm)
also have "... = (\sum_{i \leq n.} x \times S \times i * S y (n - i)) + (\sum_{i \leq n.} S \times i * (y * S y (n - i)))" by (simp add: ac_simps)
also have "... = (\sum_{i \leq n.} real (Suc i) *\text{R} (S \times (Suc i) * S y (n - i))) + (\sum_{i \leq n.} real (Suc n - i) *\text{R} (S \times i * S y (Suc n - i)))" by (simp add: times_S Suc_diff_le)
also have "(\sum_{i \leq n.} real (Suc i) *\text{R} (S \times (Suc i) * S y (n - i))) = (\sum_{i \leq \text{Suc n.} \text{ real i} *\text{R} (S \times i * S y (Suc n - i)))" by (subst sum.atMost_Suc_shift) simp
also have "(\sum_{i \leq \text{Suc n.} \text{ real (Suc n - i) *\text{R} (S \times i * S y (Suc n - i)))} = (\sum_{i \leq \text{Suc n.} \text{ real (Suc n - i) *\text{R} (S \times i * S y (Suc n - i)))} by simp
also have "(\sum_{i \leq \text{Suc n.} \text{ real i *\text{R} (S \times i * S y (Suc n - i))) + (\sum_{i \leq \text{Suc n.} \text{ real (Suc n - i) *\text{R} (S \times i * S y (Suc n - i)))} = (\sum_{i \leq \text{Suc n.} \text{ real (Suc n) *\text{R} (S \times i * S y (Suc n - i)))} by (simp flip: sum.distrib scaleR.add_left_of_nat_add)
also have "... = real (Suc n) *\text{R} (\sum_{i \leq \text{Suc n.} \text{ S \times i * S y (Suc n - i)})" by (simp only: scaleR_right.sum)
finally show "S (x + y) (Suc n) = (\sum_{i \leq \text{Suc n.} \text{ S \times i * S y (Suc n - i)})" by (simp del: sum.cl_ivl_Suc)
Painful, yet the steps of that proof are routine!

the distributive law \((x + y)z = xz + yz\)

the distributive law \(x \sum_{i \leq n} a_n = \sum_{i \leq n} x a_n\)

the distributive law \(\sum_{i \leq n} (a_n + b_n) = \sum_{i \leq n} a_n + \sum_{i \leq n} b_n\)

Shifting the index of summation and deleting a zero term

Change-of-variables is also common in such proofs

Can’t at least some of these steps be learned from similar previous proofs?
Isabelle timeline (36 years!)

1986: higher-order unification
1988: classical reasoning
1989: logical framework
1989: term rewriting simplifier
1991: polymorphism and HOL
1995: set theory libraries
1996: verification case studies
1997: axiomatic type classes
1998: classical reasoner “blast”
1999: modules for structured specifications, “locales”

2002: structured proofs: Isar
2004: Archive of Formal Proofs
2007: sledgehammer
2008: multithreading
2011: counterexample finding (nitpick and quickcheck)
2013: code generation
2015: jEdit-based prover IDE
2016: HOL Light analysis library
2017+: advanced mathematics