Interactive Theorem Proving

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

July, 2011

Interactive theorem proving

Formal verification involves using formal methods to verify correctness, for example:

- verifying that a circuit description, an algorithm, or a network or security protocol meets its specification; or
- verifying that a mathematical statement is true.

"Interactive theorem proving" is one important approach. (Model checking is another.)

Working with a proof assistant involves conveying enough information to the system to confirm that there is a formal axiomatic proof.

In fact, most proof systems actually construct a formal proof object, a complex piece of data that can be verified independently.

Interactive theorem proving

Some achievements to date:

- The four-color theorem (Gonthier, 2004)
- The prime number theorem (Avigad et al., 2004; Harrison, 2008)
- The Jordan curve theorem (Hales, 2005; Kornilowicz, 2005)
- The Gödel incompleteness theorem (Shankar, 1986; O'Connor, 2004; Harrison, 2005)
- Dirichlet's theorem (Harrison, 2009)
- Complex analysis; Cartan fixed point theorems (Harrison; Ciolli, Gentili, Maggesi, 2009)
- Measure theory, including Radon-Nikodym and entropy (various authors)

. . . and much more; see the *Journal of Automated Reasoning*, *Journal of Formalised Reasoning*, *Journal of Formalized Mathematics*, *Interactive Theorem Proving* (conference, previously TPHOLs), and Freek Wiedijk's list of 100 theorems.

Interactive theorem proving

Thomas Hales is heading the *Flyspeck* project to verify a proof of the Kepler conjecture.

- His 1998 proof involved three essential uses of computation (enumerating tame graphs, solving linear programs, reducing nonlinear constraints to linear ones).
- The formalization has led to even stronger results in discrete geometry.

Georges Gonthier is heading a project to verify the Feit-Thompson theorem:

- The original 1963 journal publication ran 255 pages.
- The formalization is constructive.
- The development includes libraries for finite group theory, linear algebra, and representation theory.

Interactive theorem proving

Vladimir Voevodsky has launched a project to develop "univalent foundations" for algebraic topology.

- Observation: constructive dependent type theory has natural homotopy-theoretic interpretations (types are spaces, equality is homotopy equivalence)
- Rules for identity types characterize homotopy theories abstractly (Quillen model categories)
- One can consistency add an axiom to the effect that "isomorphic structures are identical."

Interactive theorem proving

Despite these successes, interactive theorem proving is not "ready for prime time."

- There is a steep learning curve (it helps to have background in logic, functional programming, and/or type theory).
- Verifying even very straightforward facts can be time consuming and painful.

Two questions:

- Why should mathematicians care?
- Why should logicians care?

Interactive theorem proving

Post claimed that the undecidability of arithmetic shows that

> *. . . mathematical thinking is, and must be, essentially creative . . .*

But do we need to appeal to undecidability?

Suppose we had an oracle that could decide the truth of any mathematical statement.

How would mathematics change?

Interactive theorem proving

What we care about:

- We want our theories to be powerful, and useful.
- We want our definitions and concepts to be natural.
- We want our theorems to be interesting.
- We enjoy finding connections between different domains.
- We enjoy novel ideas and insights.
- We want to explain mathematical phenomena.
- We want to understand why theorems are true.
- We want our theorems and proofs to be correct.

Formal verification addresses only this last point.

So why bother?

Interactive theorem proving

Mathematics is distinguished by the rigorous standards for justifying its assertions.

- We try very hard not to make mistakes.
- We invest a lot of effort in refereeing.
- Even slight inaccuracies in statements in proofs and lemmas can be a source of frustration.
- Proofs are getting more and more complicated.
- Many proofs now rely on extensive computation.
- Computer algebra systems are unsound.

See Melvyn B. Nathanson's "Desperately seeking mathematical truth" in the August 2008 *Notices* of the AMS.

Interactive theorem proving

Interactive theorem proving is a technical tool, like mathematical typesetting, that can help support mathematics.

It also goes hand in hand with symbolic computation and computational support for the representation, storage, communication, and discovery of mathematical knowledge.

In the long run, such technology will have an impact on everyday mathematics.

Why should logicians care?

Interactive theorem proving

The field of formal verification explores mathematical, logical, computational, philosophical, and conceptual questions that are interesting and important in their own right.

We need to better understand:

- mathematical language (assertions, proofs)
- the way mathematical knowledge is structured
- the way mathematical concepts are used
- the semantics of mathematical and symbolic computation
- The limits of computational reasoning; where (and how) creativity sets in

Overriding themes:

- Current successes rely crucially on twentieth century advances in mathematical logic.
- Continued success will require more theoretical advances.

Outline

- Interactive theorem proving
- Contemporary systems
  - Formal frameworks
  - Proof languages
- Automated reasoning
  - Domain general search procedures
  - Decision procedures
  - Combination procedures
- Type theory and type inference
- Mathematical knowledge management
- A case study: Euclidean geometry

Formal Frameworks

Contemporary proof systems rely on a variety of frameworks:

- set theory (Mizar)
- simple type theory (HOL, HOL light, Isabelle, . . .)
- (constructive) dependent type theory (Coq)

Conisder the statement "there are no infinite descending sequences of natural numbers."

In set theory, one might render this as follows:

$$\forall f\ (f \in (\mathbb{N} \to \mathbb{N}) \to \exists i\ (i \in \mathbb{N} \wedge (f(i+1) \geq f(i)))$$

In simple type theory, there are base types like `nat` and `bool`, and type constructors like $\sigma \to \tau$, $\sigma \times \tau$, *list* σ, etc.

$$\forall f : \mathbb{N} \to \mathbb{N}\ \ \exists i : \mathbb{N}\ \ (f(i+1) \geq f(i)).$$

This gives type inference, overloading, pattern matching.

Formal Frameworks

Here is Hales' statement of the Jordan curve theorem in HOL light:

```
!C. simple_closed_curve top2 C ==>
  (?A B. top2 A /\ top2 B /\
    connected top2 A /\ connected top2 B /\
  ~(A = EMPTY) /\ ~(B = EMPTY) /\
   (A INTER B = EMPTY) /\ (A INTER C = EMPTY) /\
       (B INTER C = EMPTY) /\
      (A UNION B UNION C = euclid 2)
```

Here is a statement of the prime number theorem:

```
 (%x. pi x * ln (real x) / (real x)) ----> 1
```

Proof Languages

Computational proof assistants are supposed to verify the existence of a formal axiomatic proof, ideally providing a communicable "proof object."

There are at least two distinct styles of entering formal "proof scripts."

LCF framework: developed by Robin Milner to implement Dana Scott's "Logic of Computable Functions"; now used by the HOL theorem provers, Coq, Isabelle, ... Apply tactics to reduce goals to simpler ones.

Mizar: designed by Andrzej Trybulec, designed to model ordinary proof language. A similar language, "Isar," has been implemented by Makarius Wenzel in Isabelle.

Proof Languages

Think of an ordinary proof as a high-level description of, or recipe for constructing, a fully detailed axiomatic proof.

In formal verification, it is common to refer to proofs as "code."

```
lemma prime_factor_nat: "n ~= (1::nat) ==>
    EX p. prime p & p dvd n"
  apply (induct n rule: nat_less_induct)
  apply (case_tac "n = 0")
  using two_is_prime_nat apply blast
  apply (case_tac "prime n")
  apply blast
  apply (subgoal_tac "n > 1")
  apply (frule (1) not_prime_eq_prod_nat)
  apply (auto intro: dvd_mult dvd_mult2)
done
```

Proof Languages

```
proof (induct n rule: less_induct_nat)
  fix n :: nat
  assume "n ~= 1" and
    ih: "ALL m < n. m ~= 1 --> (EX p. prime p & p dvd m)"
  then show "EX p. prime p & p dvd n"
  proof -
  { assume "n = 0"
    moreover note two_is_prime_nat
    ultimately have ?thesis by auto }
  moreover
  { assume "prime n" then have ?thesis by auto }
  moreover
  { assume "n ~= 0" and "~prime n"
    with `n ~= 1` have "n > 1" by auto
    with `~prime n` and not_prime_eq_prod_nat obtain m k where
      "n = m * k" and "1 < m" and "m < n" by blast
    with ih obtain p where "prime p" and "p dvd m" by blast
    with `n = m * k` have ?thesis by auto }
  ultimately show ?thesis by blast
```

Proof Languages

```
Theorem Burnside_normal_complement :
  'N_G(S) \subset 'C(S) -> 'O_p^'(G) ><| S = G.
Proof.
move=> cSN; set K := 'O_p^'(G); have [sSG pS _] := and3P sylS.
have [p'K]:  p^'.-group K /\ K <| G by rewrite pcore_pgroup pcore_normal.
case/andP=> sKG nKG; have{nKG} nKS := subset_trans sSG nKG.
have{pS p'K} tiKS: K :&: S = 1 by rewrite setIC coprime_TIg ?(pnat_coprime pS).
suffices{tiKS nKS} hallK: p^'.-Hall(G) K.
  rewrite sdprodE //= -/K; apply/eqP; rewrite eqEcard ?mul_subG //=.
  by rewrite TI_cardMg //= (card_Hall sylS) (card_Hall hallK) mulnC partnC.
pose G' := G^`(1); have nsG'G : G' <| G by rewrite der_normal.
suffices{K sKG} p'G': p^'.-group G'.
  have nsG'K: G' <| K by rewrite (normalS _ sKG) ?pcore_max.
  rewrite -(pquotient_pHall p'G') -?pquotient_pcore //= -/G'.
  by rewrite nilpotent_pcore_Hall ?abelian_nil ?der_abelian.
suffices{nsG'G} tiSG': S :&: G' = 1.
  have sylG'S : p.-Sylow(G') (G' :&: S) by rewrite (pSylow_normalI _ sylS).
  rewrite /pgroup -[#|_|](partnC p) ?cardG_gt0 // -{sylG'S}(card_Hall sylG'S).
  by rewrite /= setIC tiSG' cards1 mul1n pnat_part.
apply/trivgP; rewrite /= focal_subgroup_gen ?(p_Sylow sylS) // gen_subG.
apply/subsetP=> z; case/imset2P=> x u Sx; case/setIdP=> Gu Sxu ->{z}.
have cSS: forall y, y \in S -> S \subset 'C_G[y].
  move=> y; rewrite subsetI sSG -cent_set1 centsC sub1set; apply: subsetP.
  by apply: subset_trans cSN; rewrite subsetI sSG normG.
have{cSS} [v]: exists2 v, v \in 'C_G[x ^ u | 'J] & S :=: S :^ u :^ v.
```

Automated reasoning

Most systems employ automated techniques to fill in small gaps in reasoning.

One can distinguish between:

- Domain-general methods and domain-specific methods
- Search procedures and decision procedures
- "Principled" methods and heuristics

Automated reasoning

Domain-general methods:

- Propositional theorem proving
- First-order theorem proving
- Higher-order theorem proving
- Equality reasoning
- Nelson-Oppen "combination" methods

Domain-specific methods:

- Linear arithmetic (integer, real, or mixed)
- Nonlinear real arithmetic (real closed fields, transcendental functions)
- Algebraic methods (such as Gröbner bases)

Automated methods do especially well on large, homogeneous problems; but often fail to capture even the most straightforward mathematical inferences.

Automated reasoning

General approaches to theorem proving:

- global / top-down (e.g. tableaux): goal directed, works backwards to construct a proof (or countermodel)
- local / bottom-up (e.g. resolution): start with a set of facts, reason forwards to derive additional facts

Automated reasoning

Tableau search: consider sets for formulas $\{\varphi_1, \ldots, \varphi_n\}$, read disjunctively.

Some rules for working backwards:

$$\frac{\Gamma, \varphi \qquad \Gamma, \psi}{\Gamma, \varphi \wedge \psi} \qquad \frac{\Gamma, \varphi, \psi}{\Gamma, \varphi \vee \psi} \qquad \frac{\Gamma, \varphi(a)}{\Gamma, \forall x\, \varphi(x)}$$

What about the existential quantifier?

$$\frac{\Gamma, \exists x\, \varphi(x), \varphi(?t(a, b, c, \ldots))}{\Gamma, \exists x\, \varphi(x)}$$

Notes:

- $?t$ can be instantiated to any term involving the other parameters.
- It's best to delay the choice.
- More than one term may be needed.
- All the background knowledge is lumped into Γ.

Unification

Suppose you know that for every x and y,
$(A(x, f(x, y)) \rightarrow B(x, y))$.

Suppose you also know that for every w and z, $A(g(w), z)$.

Then you can conclude $B(g(w), y)$ by solving $x = g(w)$ and
$z = f(g(w), y)$.

Theorem (Robinson)

There is an algorithm that determines whether a set of pairs $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ *of first-order terms has a unifier, and, if it does, finds a most general unifier.*

Skolemization

If φ is the formula $\forall x\ \exists y\ \forall z\ \exists w\ \theta(x, y, z, w, u)$, the *Skolem normal form* φ^S is the formula

$$\forall x, z\ \theta(x, g(x, u), z, f(x, g(x, u), z, u), u)$$

Dually, the Herbrand normal form ψ^H of ψ replaces the universal quantifiers.

- $\vdash \varphi^S \to \varphi$
- If $\varphi^S \vdash \alpha$ then $\varphi \vdash \alpha$.
- $\vdash \psi \to \psi^H$
- If $\Delta \vdash \psi^H$ then $\Delta \vdash \psi$.

Resolution

Putting it all together: $T \vdash \varphi$ if and only if $T^H \vdash \varphi^S$.

T^H is universal, and φ^S is existential. By Herbrand's theorem, $T^H \vdash \varphi^S$ if and only if there is a *propositional proof* of a disjunction of instances of φ^S from instances of T^H.

Resolution tries to prove $\bot$ from $T^H \cup \{\neg\varphi^S\}$:

- Leave the universal quantifiers implicit.
- Put all formulas in conjunctive normal form, and split up conjuncts.
- So, the goal is to prove $\bot$ from *clauses*, i.e. disjunctions of atomic formulas and literals.
- Use the resolution rule:

$$\frac{\Gamma \vee \varphi \qquad \Delta \vee \neg\varphi}{\Gamma \vee \Delta}$$

  More generally, use unification to instantiate clauses to the form above.

Resolution

Main loop:

1. Use resolution to generate new clauses.
2. Check for redundancies (subsumption) and delete clauses.

Issues:

- How much effort to put into each phase?
- How to choose new clause (biggest, widest, heaviest, . . .)?
- How to handle equality? (Paramodulation, superposition)
- How to handle other equivalence relations, transitive relations?
- How to distinguish different kinds of information (like sort information)?
- How to incorporate domain specific information, like arithmetic, or AC operations?

There are endless variations and methods of restricting the search and maintaining completeness.

Research is largely empirical (using e.g. TPTP database).

Equality reasoning

It is reasonable to simplify terms:

- $x + 0 = x$
- $x > 0 \to |x| = x$
- $y \neq 0 \to (x/y) * y = x$
- $x + (z + (y + 0) + x) = x + x + y + z$

The Knuth-Bendix completion algorithm tries to extend a set of equations to a normalizing and confluent rewrite system.

Three strategies:

- Use a simplifier.
- Build equality reasoning into other procedures.
- Use unification up to equivalence.

See Baader and Nipkow, *Term rewriting and all that*.

Higher-order unification

Sometimes mathematics requires higher-order unification:

$$P(0) \wedge \forall x\ (P(x) \to P(x+1)) \to \forall x\ P(x)$$

or

$$\sum_{x \in A}(f(x) + g(x)) = \sum_{x \in A} f(x) + \sum_{x \in A} g(x)$$

Notes:

- Second-order unification is undecidable (Goldfarb).
- Huet's algorithm is complete.
- Miller patterns are a decidable fragment.

Decision procedures

Full first-order theory:

- Quantifier elimination (integer / linear arithmetic, *RCF*, *ACF*)
- "Global methods" (Cooper, CAD)
- Reductions to Rabin's *S2S*
- Feferman-Vaught (product structures)

Sometimes it is enough to focus on the universal fragment:

- Some theories are only decidable at this level (e.g. uninterpreted functions)
- Can be more efficient (integer / linear arithmetic).
- Can use certificates.
- A lot of mathematical reasoning is close to quantifier-free.
- We'll see later: these can be *combined*.

Quantifier elimination

Theorem. The theory of $(\mathbb{R}, 0, 1, +, <)$ has quantifier-elimination, and so is decidable.

Proof. It suffices to show that if φ is quantifier-free, $\exists x\, \varphi$ is equivalent to a quantifier-free formula.

Notes:

- Can put φ in disjunctive normal form.
- $\exists x\, (\theta \vee \eta)$ is equivalent to $\exists x\, \theta \vee \exists x\, \eta$.
- $s \neq t$ is equivalent to $s < t \vee t < s$.
- $s \not< t$ is equivalent to $t < s \vee s = t$.

So, it suffices to assume φ is a conjunction of equalities and strict inequalities.

Decision procedures

Expressions that don't involve x can be brought outside the existential quantifier.

Using rational coefficients, can put expressions involving x in *pivot form*:

- $x = s$
- $x < s$
- $s < x$

φ is a conjunction of these.

If any conjunct has the form $x = s$, $\exists x\, \varphi(x)$ is equivalent to $\varphi(s)$, and we're done.

Decision procedures

Otherwise, φ is a conjunction of formulas of the form $s_i < x$ and $x < t_j$.

It is not hard to check that $\exists x\, \varphi$ is equivalent to

$$\bigwedge_{i,j} s_i < t_j.$$

Notes:

- Can allow multiplicative coefficients from any computable field.
- There are much more efficient procedures, but Fourier-Motzkin works well in practice.

Decision procedures

Other theories that are decidable:

- Presburger arithmetic
- Real closed fields
- Algebraically closed fields
- Mixed integer-linear arithmetic
- Real vector spaces, inner product spaces (Solovay, Arthan, Harrison)
- Boolean Algebra with Presburger Arithmetic (and cardinality) (Kuncak)

All have been implemented in theorem provers.

Decision procedures for universal fragments

- Integer / linear arithmetic reduces to integer / linear programming
- Real closed fields: semidefinite optimization (Parillo, Harrison)
- Algebraically closed fields: Groebner bases
- Geometry: Wu's method.
- Uninterpreted functions (congruence closure)
- Theories of arrays, lists, and so on.
- Normed spaces (Solovay, Arthan, Harrison)

Solovay et al. also show that the Π_2 fragment of the theory of metric spaces is decidable.

Harrison has also shown that Groebner bases provide a useful partial decision procedure for Π_2 statements on the integers.

Decision procedures

The bad news:

- Decision procedures are often infeasible.
- Undecidability sets in quickly.
- Ordinary mathematical reasoning is not homogeneous.

Some alternatives:

- Go back to domain general provers.
- Design heuristic procedures that use domain-specific information.
- Use "satisfiability modulo theories" technology to combine decision procedures.

Combining decision procedures

Theorem (Nelson-Oppen)

Suppose T_1 and T_2 are "stably infinite" and decidable. Suppose that the languages are disjoint, except for the equality symbol. Then the universal fragment of $T_1 \cup T_2$ is decidable.

In particular, if T_1 and T_2 have only infinite models, they are stably infinite.

This allows you to design decision procedures for individual theories and then put them together.

With additional hypotheses on the source theories, the decision procedures can be made efficient (Nelson-Oppen, Shostak, . . .).

Combining decision procedures

First idea: one can "separate variables" in universal formulas.

That is, $\forall \vec{x}\, \varphi(\vec{x})$ is equivalent to $\forall \vec{y}\, (\varphi_1(\vec{y}) \vee \varphi_2(\vec{y}))$, where φ_1 is in the language of T_1, and φ_2 is in the language of T_2.

To do this, just introduce new variables to name subterms.

Second idea: the Craig interpolation theorem.

Theorem

Suppose ψ_1 is a sentence in L_1 and ψ_2 is a sentence in L_2, such that $\vdash \psi_1 \to \psi_2$. Then there is a sentence θ in $L_1 \cap L_2$ such that

- $\vdash \psi_1 \to \theta$
- $\vdash \theta \to \psi_2$

Combining decision procedures

Let φ be any universal sentence, equivalent to $\forall \vec{x}\, (\varphi_1(\vec{x}) \vee \varphi_2(\vec{x}))$.

Then $T_1 \cup T_2 \vdash \varphi$ if and only if there is θ in the common language, such that

- $T_1 \cup \{\neg\varphi_1(\vec{x})\} \vdash \theta(\vec{x})$
- $T_2 \cup \{\neg\varphi_2(\vec{x})\} \vdash \neg\theta(\vec{x})$

We can assume θ is in disjunctive normal form. All that each disjunct can do is declare certain variables equal to one another, and others unequal!

Use the decision procedures for T_1 and T_2 to test each possibility.

Combining decision procedures

Nelson-Oppen methods are based on this idea.

- A fast propositional SAT solver "core" tries to build a satisfying assignment.
- Individual decision procedures examine proposals, and report conflicts.
- The SAT solver incorporates this information into the search.
- Some systems go beyond the universal fragment, for example, instantiating universal axioms in sensible ways.

See SMT-LIB and SMT-COMP.

There are currently (still somewhat experimental) interfaces between interactive theorem provers and SAT solvers (which can, in principle, return certificates).

There has also been some reasearch on combining theories with nontrivial overlaps. For example, Harvey Friedman and I have considered what happens when one combines linear and multiplicative fragments of RCF.

Automated reasoning

Contemporary successes build on core results in logic:

- formal axiomatic systems (equational, first-order, higher-order), semantics, and completness proofs
- cut elimination and cut-free provability
- normalization and lambda terms
- Skolemization and properties
- decision procedures (arithmetic, real closed fields, algebraically closed fields)
- model theory of algebraic structures
- Craig's interpolation lemma

The theory is a long way from useful implementation; computer scientists have produced very many novel and important ideas in that respect.

But the theory frames the whole enterprise.

Automated reasoning

Logicians really like decision procedures. But:

- A decision procedure that runs too slowly on the examples you care about is worthless.
- An unprincipled hack that gets all your inferences is fine.
- Some undecidability results rely on coding and contrived examples that never come up in practice.
- The set of inferences that can be verified in ZFC with less than 10^{100} symbols is decidable (in constant time!).

First-order and SMT frameworks are domain general, but in particular domains, there is additional information and structure.

Challenge: design partial decision procedures, or heuristic search procedures, that work well in practice.

Conceptual problem: develop a theoretical framework to make sense of the last phrase.

Decidability and completeness are a poor proxy.

Automated reasoning

There is a tension between domain general methods and domain specific methods. What we need are general approaches to domain specific reasoning (e.g. with domain specific features encoded by specific rules and parameters).

One wants transparency: one should have a sense of when the methods should succeed, and when they fail, it should be possible to determine why (using traces, or "binary" checks).

One wants flexibility to get things working again (adding local information, setting parameters, adjusting behavior based on context).

One wants efforts to scale.

Automated reasoning

Further speculation:

- "Guided" forward reasoning seems promising, especially if one can limit the data gathered (type information, set inclusions, inequalities, relationships in a diagram, etc.).
- Cooperation between specialized modules seems important.
- A lot more experimentation is needed, with real mathematical contexts.
- We also need a better theory, to characterize the situations in which one can expect good behavior.

This provides good opportunities for collaborations between logicians, mathematicians, and computer scientists (and philosophers).

Outline

- Interactive theorem proving
- Contemporary systems
  - Formal frameworks
  - Proof languages
- Automated reasoning
  - Domain general search procedures
  - Decision procedures
  - Combination procedures
- Type theory and type inference
- Mathematical knowledge management
- A case study: Euclidean geometry

Type inference

Georges Gonthier's *Mathematical components* group has been working on a formal verification of the Feit-Thompson theorem.

This involves finite group theory, linear algebra, and representation theory.

I spent a sabbatical year (2009-2010) working on the project.

Here are some things I learned.

Type inference

Consider the following mathematical statements:

"For every $x \in \mathbb{R}$, $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$."

"If G and H are groups and f is a homomorphism from G to H, then for every $a, b \in G$, $f(ab) = f(a)f(b)$."

"If F is a field of characteristic p and $a, b \in F$, then $(a+b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p$."

How do we parse these?

Type inference

Observations:

1. The index of the summation is over the natural numbers.
2. $\mathbb{N}$ is embedded in $\mathbb{R}$.
3. In "$a \in G$," G really means the underlying set.
4. ab means multiplication in the relevant group.
5. p is a natural number (in fact, a prime).
6. The summation operator make sense for any monoid (written additively).
7. The summation enjoys extra properties if the monoid is commutative.
8. The additive part of any field is so.
9. $\mathbb{N}$ is also embedded in any field.
10. Alternatively, any abelian group is a $\mathbb{Z}$-module, etc.

Type inference

Spelling out these details formally can be painful.

Typically, the relevant information can be *inferred* by keeping track of the *type* of objects we are dealing with:

- In "$a \in G$," the "$\in$" symbol expects a set on the right.
- In "ab," multiplication takes place in "the" group that a is assumed to be an element of.
- In "$x^i/i!$," one expects the arguments to be elements of the same structure.

Type inference: not only inferring types, but also inferring information from type considerations.

Type inference

Structure hierarchies:

- Subclasses: every abelian group is a group
- Reducts: the additive part of a ring is an abelian group
- Instances: the integers are an abelian group
- Embedding: the integers are embedded in the reals
- Uniform constructions: the automorphisms of a field form a group

Advantages:

- Reusing notation: 0, $+$, $a \cdot b$
- Reusing definitions: $\sum_{i \in I} a_i$
- Reusing facts: identities involving sums

Type inference

Observations:

- Type inference occurs when one parses an expression, but also when one applies a lemma.
- The goal is to omit information systematically.
- There are really two kinds of information that is omitted:
  - data: the relevant group multiplication, the relevant embedding
  - facts: the fact that an operation is associative, the fact that a set is closed under an operation
- Under the Curry-Howard isomorphism, facts and data look the same.
- A good deal of technology is imported from the theory of programming languages (but there are differences).
- There is no sharp line between "type" information and genuinely mathematical information. (Consider $\binom{n}{k}$.)

Type inference

System	Framework	Type inference
Isabelle	Simple type theory	Axiomatic type classes
Mizar	Set theory	Soft typing
Coq	Dependent type theory	Canonical structures or Type classes, etc.

Features of Coq:

- It is based on a expressive dependent type theory.
- The underlying logic is constructive.
- Every term has a computational interpretation.
- Type checking is, in principle, decidable.
- For that reason, it is also rigid.

Type inference in Coq

Mechanisms for type inference in Coq:

- *Implicit arguments*: one can omit arguments that can be inferred from a dependent type
- *Coercions*: cast objects to different types
- *Canonical structures*: can view a particular structure as an instance of a class

In addition, Coq's type inference engine makes use of the computational interpretation, e.g. expanding definitions and simplifying terms as necessary.

Dependent types

Sometimes one wants a type to depend on *parameters*.

Examples:

- `list A n`: lists of elements of type A of length n
- `Zmod n`: the integers modulo n (say, as a ring)
- `Rvec n`: the vector space $\mathbb{R}^n$

The type constructor $A \to B$ is generalized to a *dependent product*, $\prod_{x \in A} B\ x$.

In Coq, this is written `forall(x : A)Bx`.

The type constructor $A \times B$ is generalized to a *dependent sum*, $\sum_{x \in A} B\ x$.

Dependent types

```
Record group : Type := Group
{
  carrier : Type;
  mulg : carrier -> carrier -> carrier;
  oneg : carrier;
  invg : carrier -> carrier;
  mulgA : associative mulg;
  ...
}
```

The components of `G : group` are `carrier G`, `mulg G`, ...

Given `g, h : carrier G`, we have `mulg G g h : carrier G`.

So `mulg` has type `forall (G : group), carrier G -> carrier G -> carrier G.`

Implicit arguments and coercions

One can also write `mulg _ g h`, leaving the first argument *implicit*.

Type inference has to solve `carrier ? = carrier G`, which is easy.

```
Notation "g * h" := (mulg _ g h).
```

Now one can write `g * h` for group multiplication.

One can also define

```
Coercion carrier : group >-> Type.
```

Then `g : G` is interpreted as `g : carrier G`.

Canonical structures

Suppose we define

```
IntGroup := Group int addi zeroi negi addiA ...
```

Given `i,j : int`, this will let us (perversely) write `mulg IntGroup i j` for `i + j`, and (less perverseley) apply facts about groups.

What happens if write `i * j`?

Type inference has to solve `carrier ? = int`, and gets stuck.

Declaring

```
Canonical Structure IntGroup.
```

registers the hint `carrier IntGroup = int` for use in type inference.

Summary / recap

Type checking is triggered when:

- parsing an expression
- applying a lemma

Often implicit arguments or facts need to be inferred.

Mechanisms:

- Unification: pattern matching to infer implicit arguments.
- Coercions: cast objects to different types
- Canonical structures: register unification hints that associate structures with instances
- Unfolding definitions, simplifying terms

Finite group library

In the finite group library, type inference is used in a number of ways:

- To recognize when structures have decidable equality and choice functions, satisfy extensionality, and so on.
- To define "big operations" such as $\sum$, $\prod$, $\bigcap$, $\bigcup$, $\bigwedge$, $\bigvee$
- To mediate between sets and structures (e.g. $G \cap H$ and $C_G(A)$ act as both sets and groups).
- To manage class inclusions (rings, commutative rings, fields)
- To manage algebraic constructions (matrices over a ring, polynomials over a ring, quotient groups)
- To infer views (e.g. abelian group as a $\mathbb{Z}$-module)
- The mediate between functions and morphisms
- To view both predicates and lists as sets (e.g. Px vs. $x \in P$).

Examples

```
Lemma commg_subl : forall G H,
   ([~: G, H] \subset G) = (H \subset 'N(G)).

Lemma nilpotent_proper_norm : forall G H,
  nilpotent G -> H \proper G -> H \proper 'N_G(H).

Lemma morphim_center : forall rT A D
  (f : {morphism D >-> rT}),
    f @* 'Z(A) \subset 'Z(f @* A).

Lemma quotient_cents2 : forall A B K,
    A \subset 'N(K) -> B \subset 'N(K) ->
  (A / K \subset 'C(B / K)) = ([~: A, B] \subset K).
```

Examples

```
Theorem Sylow's_theorem :
  [/\ forall P,
      [max P | p.-subgroup(G) P] = p.-Sylow(G) P,
      [transitive G, on 'Syl_p(G) | 'JG],
      forall P, p.-Sylow(G) P ->
        #|'Syl_p(G)| = #|G : 'N_G(P)|
   &  prime p -> #|'Syl_p(G)| %% p = 1%N].

Lemma card_GL : forall n, n > 0 ->
  #|'GL_n[F]| = (#|F| ^ 'C(n, 2) *
    \prod_(1 <= i < n.+1) (#|F| ^ i - 1))%N.

Theorem Cayley_Hamilton : forall A,
  (Zpoly (char_poly A)).[A] = 0.
```

Mathematical Knowledge Management

Suppose one proves a theorem in one theorem prover. Can one import it into another?

Problems:

- different foundational frameworks
- different axiomatic bases
- different definitions (e.g. reals as Cauchy sequences vs. Dedekind cuts.)
- different side conditions (e.g. $x/0$.)
- proof terms get really big

Mathematical Knowledge Management

How can we decide which theorems from the library should be sent to a resolution theorem prover?

Machine learning methods: Larry Paulson, Josef Urban.

How do we register the fact that certain facts are meant to be used in certain ways?

- $x + 0 = x$ is a simplification rule.
- $(A = B) \leftrightarrow \forall x\, (x \in A \leftrightarrow x \in B)$ is the canonical way to prove set equality in some contexts, but *not others* (such as finite group theory).

Being able to store, find, and exchange mathematical information is important. We just don't know how to do it.

Mathematical Knowledge Management

How can we reason about and verify mathematical computation?

Verifying code is a big industry.

But mathematical code is a very special case.

Reflection is an important mechanism, allowing one to internalize computation.

Outline

- Interactive theorem proving
- Contemporary systems
  - Formal frameworks
  - Proof languages
- Automated reasoning
  - Domain general search procedures
  - Decision procedures
  - Combination procedures
- Type theory and type inference
- Mathematical knowledge management
- A case study: Euclidean geometry

References

Google around to find tutorials and introductions to the major theorem provers: Isabelle, Coq, HOL light, Mizar...

There are also web pages for the major projects:

- Gonthier's "Mathematical components"
- Hales' "Flyspeck"
- Voevodsky, Homotopy type theory (HoTT)

For automated reasoning and decision procedures:

- John Harrison, *Handbook of Practical Logic and Automated Reasoning*

Conclusions

Remember the main message:

- Successes in formal verification are based on the deep understanding of mathematical deductive reasoning that mathematical logic provides.
- There are still a number of fundamental conceptual problems that need to be addressed.
- We need entirely new logical theories to address them.