Proof Systems in Computer Science

Jeremy Avigad

Department of Philosophy
Department of Mathematical Sciences

Hoskinson Center for Formal Mathematics

Carnegie Mellon University

September 20, 2022

Proof systems in computer science

The title of this workshop:

Proofs and Formalization in Logic, Mathematics, and Philosophy

Proof systems in computer science

The title of this workshop:

Proofs and Formalization in Logic, Mathematics, and Philosophy

The title of this talk could have been:

Proofs and Formalization in Computer Science
(which should be of interest in Logic, Mathematics, and Philosophy)

Proof systems in computer science

Some areas that rely on notions of proof:

- automated reasoning and proof search
- reference checkers
- knowledge representation
- explainable AI
- cryptographic proof systems
- interactive theorem proving

Proof systems in computer science

From the Optimal Proofs project:

> *There are many forms of reasoning: mathematical, legal, fallible, rational, and countless others. Logics are the formalisation of such forms of reasoning. The quality of a logic is first and foremost determined by its description. This project aims to establish, for any given logic, what the best possible description is.*

I will explore some of the design goals in computer science.

Generalities

To a computer scientist, a proof is a structured piece of data.

Proof formats are generally intertranslatable.

The issues are generally pragmatic: various forms of efficiency, usability, utility.

Generalities

Most broadly: a proof is any sort of certificate that is easy to check, or at least easier than some alternative (like finding a proof and checking it).

A language L is in NP if and only if there are a polynomial $p(n)$ and a PTIME relation $R(x, y)$ such that for every x,

$$x \in L \iff \exists y \, (|y| < p(|x|) \wedge R(x, y)).$$

The witnesses y are “proofs of membership.”

A pair (m, k) with $m, k > 1$ and $n = m \cdot k$ is a proof that n is composite.

In convex optimization, the solution to a dual problem certifies a bound on the primal.

Automated reasoning

I'll focus on *formal methods*: logic-based methods for specification and verification of software, hardware, networks, security protocols, and complex systems.

Alternatively, we might be interested in applications of formal methods to mathematics.

Key tools:

- SAT solvers
- SMT solvers
- first-order theorem provers
- equational reasoners
- higher-order theorem provers
- model checkers

Automated reasoning

A *SAT solver* is supposed to find a satisfying assignment to a CNF formula, or determine that there aren't any.

A *DPLL* search employs:

- the splitting rule
- unit propagation
- the pure literal rule

Modern solvers use conflict-driven clause learning and lots of heuristics.

Automated reasoning

A satisfying assignment itself is a proof of satisfiability.

The record of an unsuccessful search itself can be interpreted as a proof of unsatisfiability.

One can also straightforwardly extract a resolution refutation.

Automated reasoning: design goals

The main design goal for a search calculus (and method) is efficiency.

Efficiency in practice is most important, but theory provides some guidance.

For example, even for propositional logic, there are calculi that are known to be exponentially worse than others.

See the literature on *proof complexity*. This offers various measures of efficiency for proof calculi.

Automated reasoning

SMT solvers combine decision procedures on top of a SAT engine.

Two important ones:

- linear real arithmetic (the theory of $(\mathbb{R}, 0, 1, +, \leq)$)
- linear integer arithmetic (the theory of $(\mathbb{Z}, 0, 1, +, \leq)$)

For real arithmetic, modern solvers use ideas from the simplex algorithm. (See Dutertre and de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T).")

In other words, they use domain-specific search calculi, which can be seen as domain-specific proof systems.

Automated reasoning

There are:

- equational theorem provers
- first-order theorem provers
- higher-order theorem provers
- ...

These often use finely tuned search calculi with exotic rules.

Design goals:

- soundness
- completeness (usually)
- efficiency in practice

Reference checkers

In the annual SAT competition, SAT solvers are required to output proofs of unsatisfiability in a format known as DRAT.

A central design criteria: it has to be easy for conventional solvers to output such proofs without sacrificing performance.

There are tools that will compile DRAT proofs to more efficiently-checkable formats, like LRAT.

Reference checkers

This is becoming an industry: there are proposed proof formats for SMT solvers, first-order theorem provers, QSAT solvers, model counters, etc.

Once again, in the SAT case, proof complexity provides other measures of success.

For example, Samuel R. Buss and Neil Thapen, "DRAT and propagation redundancy proofs without new variables," *Logical Methods in Computer Science*, 17(2:12), 2021.

Reference checkers: design goals

To summarize:

For a search calculus, one cares about the efficiency of the search (in theory or in practice).

For an output format, one cares about:

- ease of generation
- efficiency of checking
- length of proof (in theory, in practice)

Knowledge representation

Expert systems and natural language processing require drawing inferences from large databases efficiently.

From W3C:

> *The W3C Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be exploited by computer programs, e.g., to verify the consistency of that knowledge or to make implicit knowledge explicit.*

One needs a calculus to draw inferences. Ideally a system can provide justification.

Explainable AI

A similar issue arises with respect to data mining and machine learning.

Data science and machine learning use large databases, large models, statistical inference, and neural nets are used to draw conclusions.

There is a lot of work on “explainable AI,” where the goal is to explain and justify conclusions.

I don’t know enough about these topics to say much more.

Proof systems in computer science

Areas that rely on notions of proof, revisited:

- automated reasoning and proof search
- reference checkers
- knowledge representation
- explainable AI
- cryptographic proof systems
- interactive theorem proving

Cryptographic proof systems

In the 1990s, in computational complexity, researchers studied *interactive proof systems*.

- A resource-powerful prover is trying to convince a resource-constrained verifier of some fact.
- The protocols use random coin flips.

The verifier makes some random queries, and if the prover responds adequately, with high probability the claim is true.

Variants replace randomness with one-way functions and pseudorandom generators.

The results hold modulo cryptographic conjectures.

Cryptographic proof systems

Let $R(x, y)$ be some predicate, and let a be some public data.

My goal: convince you that $\exists y\ R(a, y)$.

One solution: I show you b, and you check $R(a, b)$. If b is long and $R(a, b)$ is hard to check, this is inefficient.

Cryptographic proof protocols are designed to do this probabilistically but more efficiently.

Cryptographic proof systems

These are now used in “layer 2” blockchain solutions.

Think of blockchain as a magic ledger in the sky.

- the ledger grows, but nothing is ever deleted
- users can own resources, like bitcoin, and transfer them.

The problem is that writing a lot of data to blockchain is expensive.

Layer 2 solutions: do as much as possible off chain.

Cryptographic proof systems: design criteria

I have been working as a consultant for StarkWare Industries.

A STARK is an IPS that is designed to

- be Scalable
- be Transparent
- provide ARguments of Knowledge

Also:

- post quantum
- zero knowledge

These are design criteria!

Cryptographic proof systems

StarkWare uses the technology to execute smart contracts efficiently.

A prover can convince those maintaining the blockchain that the execution of a computation has a claimed result, without having to carry out the computation.

Interactive theorem proving

Contemporary *proof assistants* are now used to construct complex proofs in mathematics and computer science.

Users write proofs in a regimented proof language, like a programming language.

The computer provides continuous feedback.

A small, trusted kernel ensures that there is a formal axiomatic proof of the claimed result.

Interactive theorem proving

Some systems with substantial mathematical libraries:

- Mizar (set theory)
- HOL (simple type theory)
- Isabelle (simple type theory)
- HOL Light (simple type theory)
- Coq (dependent type theory)
- ACL2 (primitive recursive arithmetic)
- PVS (set theory / dependent type theory)
- Agda (dependent type theory)
- Metamath (set theory)
- Lean (dependent type theory)

Interactive theorem proving

There are two main markets for proof assistants (not disjoint):

- Formal verification
  - Software
  - Hardware (like circuits and processors)
  - Cyber-physical systems
  - Networks and privacy
  - Financial software
- Mathematics

The first is by now well established.

Interactive theorem proving: design goals

In computer science, the key challenge is complexity:

- There are lots of components.
- We need to model and keep track of state.

There are lots of special-purpose logics and languages for reasoning about systems:

- Hoare logics
- Separation logics
- Logics for concurrency
- Dynamic logic
- Differential dynamic logic (for cyber-physical systems)

See also Why3, Iris, Dafny, F*, ...

Interactive theorem proving

There is growing enthusiasm for formal methods in mathematics:

- *Quanta:* "Building the mathematical library of the future"
- *Quanta:* "At the Math Olympiad, computers prepare to go for the gold"
- *Nature:* "Mathematicians welcome computer-assisted proof in 'grand unification' theory"
- *Quanta:* "Proof Assistant Makes Jump to Big-League Math"

Kevin Buzzard recently gave a talk, "The Rise of Formalism in Mathematics," at the 2022 International Congress of Mathematicians.

See also the recent conference, "Lean for the Curious Mathematician," at ICERM.

Interactive theorem proving

Common foundations include set theory, simple type theory, and dependent type theory.

A foundation should be

- appropriate (we believe it)
- simple (we can implement it reliably)
- expressive (we can use it comfortably)

See my article, “Foundations,” for an upcoming handbook on proof assistants.

Interactive theorem proving

What really matters isn't the choice of foundation but the infrastructure that is built on top of it.

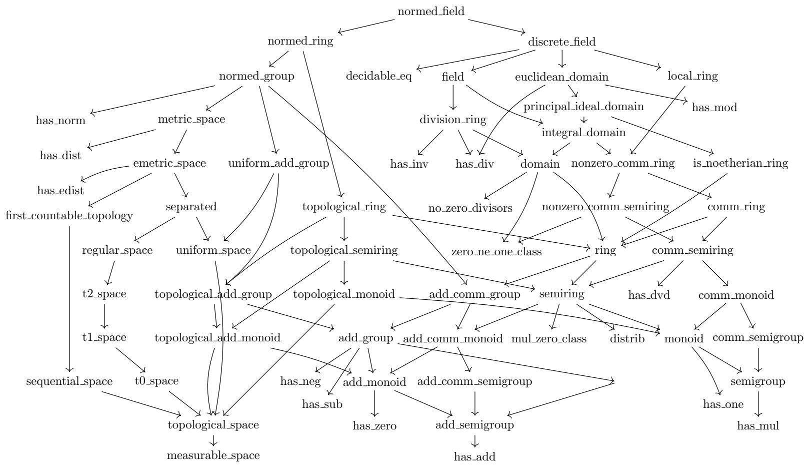
Formalization is hard and formal proofs are complicated.

We care about:

- ease and efficiency of expression
- ease and efficiency of inference
- quality of interaction.

Interactive theorem proving

In Lean's library *mathlib*, the algebraic hierarchy has hundreds of classes and thousands of instances.



Interactive theorem proving

The real numbers are simultaneously an instance of a field, an ordered field, a normed field, a metric space, a topological space, a uniform space, a vector space (over the reals), a manifold, a measure space, . . .

Think about what is needed to make sense of this:

```
variables (f g : ℝ × ℝ → ℝ)

#check f + g
#check 3 · f
#check continuous f
#check measurable f
```

Interactive theorem proving

```
@has_add.add.{0} (prod.{0 0} real real → real)
  (@pi.has_add.{0 0} (prod.{0 0} real real) (λ (α :
    prod.{0 0} real real), real)
     (λ (i : prod.{0 0} real real), real.has_add))
  f
  g :
  prod.{0 0} real real → real

@has_smul.smul.{0 0} nat (prod.{0 0} real real → real)
  (@function.has_smul.{0 0 0} (prod.{0 0} real real) nat
    real (@add_monoid.has_smul_nat.{0} real
    real.add_monoid))
  (@bit1.{0} nat nat.has_one nat.has_add
    (@has_one.one.{0} nat nat.has_one))
  f :
  prod.{0 0} real real → real
```

Interactive theorem proving

```
@continuous.{0 0} (prod.{0 0} real real) real
  (@prod.topological_space.{0 0} real real
     (@uniform_space.to_topological_space.{0} real
        (@pseudo_metric_space.to_uniform_space.{0} real
    real.pseudo_metric_space))
     (@uniform_space.to_topological_space.{0} real
        (@pseudo_metric_space.to_uniform_space.{0} real
    real.pseudo_metric_space)))
  (@uniform_space.to_topological_space.{0} real
     (@pseudo_metric_space.to_uniform_space.{0} real
    real.pseudo_metric_space))
  f

@measurable.{0 0} (prod.{0 0} real real) real
  (@prod.measurable_space.{0 0} real real
    real.measurable_space real.measurable_space)
  real.measurable_space
  f
```

Interactive theorem proving

How about these?

```
variables (f g : metric.sphere (0 : ℝ × ℝ) 3 → ℝ)

#check f + g
#check 3 • f
#check continuous f
#check measurable f
```

Interactive theorem proving

```
@has_add.add.{0}
(@coe_sort.{1 2} (set.{0} (prod.{0 0} real real)) Type (@set.has_coe_to_sort.{0} (prod.{0 0}
    real real))
   (@metric.sphere.{0} (prod.{0 0} real real)
      (@prod.pseudo_metric_space_max.{0 0} real real real.pseudo_metric_space
    real.pseudo_metric_space)
      (@has_zero.zero.{0} (prod.{0 0} real real) (@prod.has_zero.{0 0} real real
    real.has_zero real.has_zero))
      (@bit1.{0} real real.has_one real.has_add (@has_one.one.{0} real real.has_one))) →
 real)
(@pi.has_add.{0 0}
   (@coe_sort.{1 2} (set.{0} (prod.{0 0} real real)) Type (@set.has_coe_to_sort.{0} (prod.{0
    0} real real))
      (@metric.sphere.{0} (prod.{0 0} real real)
         (@prod.pseudo_metric_space_max.{0 0} real real real.pseudo_metric_space
    real.pseudo_metric_space)
         (@has_zero.zero.{0} (prod.{0 0} real real) (@prod.has_zero.{0 0} real real
    real.has_zero real.has_zero))
         (@bit1.{0} real real.has_one real.has_add (@has_one.one.{0} real real.has_one))))
   (λ
    (α :
      @coe_sort.{1 2} (set.{0} (prod.{0 0} real real)) Type (@set.has_coe_to_sort.{0}
    (prod.{0 0} real real))
        (@metric.sphere.{0} (prod.{0 0} real real)
           (@prod.pseudo_metric_space_max.{0 0} real real real.pseudo_metric_space
    real.pseudo_metric_space)
           (@has_zero.zero.{0} (prod.{0 0} real real) (@prod.has_zero.{0 0} real real
    real.has_zero real.has_zero))
           (@bit1.{0} real real.has_one real.has_add (@has_one.one.{0} real real.has_one)))),
    real)
   (λ
    (i :
      @coe_sort.{1 2} (set.{0} (prod.{0 0} real real)) Type (@set.has_coe_to_sort.{0}
    (prod.{0 0} real real))
        (@metric.sphere.{0} (prod.{0 0} real real)
```

Interactive theorem proving: design goals

The moral: everyday mathematics leaves a lot implicit.

Formalization requires making everything explicit.

We want a proof system to fill in as much information as possible.

Summing up

We have considered:

- automated reasoning and proof search
- reference checkers
- cryptographic proof systems
- interactive theorem proving

Summing up

We have considered:

- automated reasoning and proof search
  goals: efficiency of search, size of proof
- reference checkers
- cryptographic proof systems
- interactive theorem proving

Summing up

We have considered:

- automated reasoning and proof search
  goals: efficiency of search, size of proof
- reference checkers
  goals: ease of production, reliability, reusability
- cryptographic proof systems
- interactive theorem proving

Summing up

We have considered:

- automated reasoning and proof search
  goals: efficiency of search, size of proof
- reference checkers
  goals: ease of production, reliability, reusability
- cryptographic proof systems
  goals: scalability, reliability, privacy
- interactive theorem proving

Summing up

We have considered:

- automated reasoning and proof search
  goals: efficiency of search, size of proof
- reference checkers
  goals: ease of production, reliability, reusability
- cryptographic proof systems
  goals: scalability, reliability, privacy
- interactive theorem proving
  goals: expressiveness, efficiency, quality of user interaction

Summing up

I promised that these topics would be of interest to Logic, Mathematics, and Philosophy.

Summing up

I promised that these topics would be of interest to Logic, Mathematics, and Philosophy.

Where's the philosophy?

Summing up

I promised that these topics would be of interest to Logic, Mathematics, and Philosophy.

Where's the philosophy?

Think of this as applied epistemology. Proofs warrant knowledge, and can be shared and checked independently.

Sometimes it takes a lot of thought to decide what the design goals are, let alone how best to achieve them.

Theory and thoughtful reflection are important.

Summing up

If you are interested in proof systems, computer science is a good place to look:

- There are interesting mathematical, logical, and philosophical questions.
- The topics are timely, and the fields are lively.