can computers prove theorems?

luís cruz-filipe

department of mathematics and computer science
university of southern denmark

mathematical logic webinar
faculdade de ciências, universidade de lisboa
may 31st, 2021

outline

computers and mathematical proofs

the 4-color theorem

Appel, Haken and Koch (1977)

- traditionally presented as first example of a computer proof
- sparked a discussion on what a proof is

computers and mathematical proofs

the 4-color theorem

Appel, Haken and Koch (1977)

- traditionally presented as first example of a computer proof
- sparked a discussion on what a proof is

more than 10 years before...

Theorem 5. $S(7) = 16$.

Proof. This theorem was proved by exhaustive enumeration on a CDC G-21 computer at Carnegie Institute of Technology in 1966. The program was written by Mr. Richard Grove, and its running time was approximately

(Floyd & Knuth, 1973)

. . . but are these mathematical proofs?

these proofs rely on an ad-hoc program

- is the program correct?
- is the program running correctly?

. . . but are these mathematical proofs?

these proofs rely on an ad-hoc program

- is the program correct?
- is the program running correctly?

how does peer-reviewing address these "proofs"?

- trust the program?
- rerun the program?
- reimplement the program?
- or. . . ?

the Curry–Howard correspondence

what it is

correspondence between:

- in proofs a sequent calculus for some logic
- and type derivations in a particular type system

how it helps us

- propositions are types
- proofs are programs
- proof verification is type checking

$\rightsquigarrow$ type checking is often easy to implement

theorem provers based on type theory

typical features

- expressive type theory, corresponding to an expressive logic
- interactive ways to write proofs/build terms
- powerful automation techniques

theorem provers based on type theory

typical features

- expressive type theory, corresponding to an expressive logic
- interactive ways to write proofs/build terms
- powerful automation techniques

what do we need to trust?

the type checker (and nothing else)

theorem provers based on type theory

typical features

- expressive type theory, corresponding to an expressive logic
- interactive ways to write proofs/build terms
- powerful automation techniques

what do we need to trust?

the type checker (and nothing else)

de Bruijn principle

the critical part of a program used in a mathematical proof should be as small and simple as possible

the reality: a bit of everything...

formal proofs using trusted theorem provers (e.g. the 4-color theorem)

the reality: a bit of everything...

formal proofs using trusted theorem provers (e.g. the 4-color theorem)

proofs using *ad-hoc* programs with a small de Bruijn kernel

the reality: a bit of everything. . .

formal proofs using trusted theorem provers (e.g. the 4-color theorem)

proofs using *ad-hoc* programs with a small de Bruijn kernel

proofs relying on calculations by extremely complex computer algebra systems

"proofs" relying on black box systems known to be faulty

an unavoidable evolution

- the use of computers in mathematics is becoming more widespread
- computer proofs were the topic of an expert panel discussion at ICM 2018
- we should understand the different styles of proofs, and push for the "right" ones
- ... and our research can actually benefit from the right tools

outline

a combinatorial problem

the problem

find $S(n)$: length of the minimal sequence of compare-and-swap operations that sorts every input of length n

a combinatorial problem

the problem

find $S(n)$: length of the minimal sequence of compare-and-swap operations that sorts every input of length n

the good part

some theoretical properties of such sequences

a combinatorial problem

the problem

find $S(n)$: length of the minimal sequence of compare-and-swap operations that sorts every input of length n

the good part

some theoretical properties of such sequences

the bad part

no known "clever" way to solve it

- essentially: explore the search space (with some pruning)

historical evolution

Floyd & Knuth, 1973

- determine $S(2)$, $S(3)$ and $S(5)$ by hand (exhaustively)
- $S(7)$ determined by a computer program (which? how?)
- $S(4)$, $S(6)$ and $S(8)$ follow from a theoretical result

historical evolution

- Floyd & Knuth, 1973: $S(2)$–$S(8)$

Codish, Cruz-Filipe, Frank & Schneider-Kamp, 2014

- new theoretical result
- $S(9)$ determined by a computer program
- $S(10)$ follows from a theoretical result

historical evolution

- Floyd & Knuth, 1973: $S(2)$–$S(8)$
- Codish, Cruz-Filipe, Frank & Schneider-Kamp, 2014: $S(9)$ and $S(10)$

Hardis (unpublished)

- new theoretical result
- $S(11)$ determined by a computer program
- $S(12)$ follows from a theoretical result

historical evolution

- Floyd & Knuth, 1973: $S(2)$–$S(8)$
- Codish, Cruz-Filipe, Frank & Schneider-Kamp, 2014: $S(9)$ and $S(10)$
- Hardis, 2019 (unpublished): $S(11)$ and $S(12)$

for completeness...

theoretical result by Van Voorhis (1972), $S(14)$ does *not* follow from $S(13)$

a difference in approach

Floyd & Knuth, 1973

Theorem 5. $S(7) = 16$.

Proof. This theorem was proved by exhaustive enumeration on a CDC G-21 computer at Carnegie Institute of Technology in 1966. The program was written by Mr. Richard Grove, and its running time was approximately

a difference in approach

Floyd & Knuth, 1973

Theorem 5. $S(7) = 16$.

Proof. This theorem was proved by exhaustive enumeration on a CDC G-21 computer at Carnegie Institute of Technology in 1966. The program was written by Mr. Richard Grove, and its running time was approximately

Codish, Cruz-Filipe, Frank & Schneider-Kamp, 2014

computer program to explore state space: generate all successors, prune unnecessary branches, rinse and repeat

- must generate all successors
- must not prune too much

(algorithms included in publication)

but is this enough?

de Bruijn kernel

16 lines of VERY simple prolog code

- but: must also trust prolog interpreter. . .

but is this enough?

de Bruijn kernel

16 lines of VERY simple prolog code

- but: must also trust prolog interpreter. . .

a formal proof in Coq

- program produces witnesses for pruning
- checker reruns algorithm using witnesses
- checker is proved correct in Coq

but is this enough?

de Bruijn kernel

16 lines of VERY simple prolog code

- but: must also trust prolog interpreter. . .

a formal proof in Coq

- program produces witnesses for pruning
- checker reruns algorithm using witnesses
- checker is proved correct in Coq

a new tradition?

Hardis (2019) skips the implementation, and instead includes a formalization in Isabelle/HOL

outline

there's more to the story...

alternative approach

encode "$S(n) \leq k$" as a propositional formula and apply a sat-solver

$\rightsquigarrow$ big success stories of sat-solving in the last decade

there's more to the story...

alternative approach

encode "$S(n) \leq k$" as a propositional formula and apply a sat-solver

$\rightsquigarrow$ big success stories of sat-solving in the last decade

sat-solvers in a nutshell

programs that solve the propositional satisfiability problem

- based on resolution
- when that fails, case analysis
- lots of heuristics
- lots of smart, derived rules

potential issues

verifiability

- "yes": a valuation is given (can be checked independently)
- "no": that's it

Heule, 2013

seminal paper claiming verification of negative answers is unfeasible

potential issues

verifiability

- “yes”: a valuation is given (can be checked independently)
- “no”: that’s it

Heule, 2013

seminal paper claiming verification of negative answers is unfeasible

problems

- black-box complex systems, highly optimized, kept secret
- sketchy presentations in publications (at least for logicians)

potential issues

verifiability

- "yes": a valuation is given (can be checked independently)
- "no": that's it

Heule, 2013

seminal paper claiming verification of negative answers is unfeasible

the elephant in the room

the encoding

building trust

Heule and others, 2010s

formats for communicating unsatisfiability proofs

- too tailored to the individual systems
- can only be checked by the program that produced them. . .

building trust

Heule and others, 2010s

formats for communicating unsatisfiability proofs

- too tailored to the individual systems
- can only be checked by the program that produced them...

Cruz-Filipe, Marques-Silva and Schneider-Kamp, 2017

- system-independent language for certificates
- certified verifier
- able to check the largest proof (at the time): 400TB

the boolean pythagorean triples problem

problem

can we partition the natural numbers in two disjoint sets such that no set contains a pythagorean triple?

the boolean pythagorean triples problem

problem

can we partition the natural numbers in two disjoint sets such that no set contains a pythagorean triple?

answer (Heule 2016)

no!

the boolean pythagorean triples problem

problem

can we partition the natural numbers in two disjoint sets such that no set contains a pythagorean triple?

answer (Heule 2016)

no!

proof

a sat-solver said so

the boolean pythagorean triples problem

problem

can we partition the natural numbers in two disjoint sets such that no set contains a pythagorean triple?

proof technique

- encode the finite instance with the set $\{1, \ldots, 7825\}$
- generate a propositional formula (c program)
- split the formula in 1000 different cases
- solve each case with a sat-solver
- all cases unsat $\rightarrow$ theorem

do we have a mathematical proof?

lots of different issues

- prove that the encoding is correct
- prove that the splitting is correct
- prove that each unsatisfiability claim is correct

do we have a mathematical proof?

lots of different issues

- prove that the encoding is correct
- prove that the splitting is correct
- prove that each unsatisfiability claim is correct

the bad news

- all these steps have been done
- unfortunately: not much sympathy for the effort
- currently: mathematical results are still being "proved" without formal verification

outline

a problematic area

the area in a nutshell

type systems for verifying distributed programs

- equivalent to a resource logic (think linear logic)
- two layers of types/formulas (global and local)
- two layers of programs
- four different, related, systems

a problematic area

the area in a nutshell

type systems for verifying distributed programs

- equivalent to a resource logic (think linear logic)
- two layers of types/formulas (global and local)
- two layers of programs
- four different, related, systems

the challenge

conference-first publications: 16-page limit

- very compressed presentations, proofs in appendix
- very boring straightforward proofs by structural induction

disturbing signals

too many wrong proofs

- in reference books
- in published articles

disturbing signals

too many wrong proofs

- in reference books
- in published articles

Maksimovic & Schmitt, 2015

attempted formalization of a research article in Coq

- nearly all proofs were wrong
- induction hypotheses were not strong enough
- one lemma could not be proved by structural induction

our own experiment

choreographic programming

a type system with data: merges programs and their types

- only local and global view (two systems)

our own experiment

choreographic programming

a type system with data: merges programs and their types

- only local and global view (two systems)

motivation

reviewing process for a journal publication: three years

- no real “problems”
- but not enough details in the (very long and boring) proofs

Coq formalization

Cruz-Filipe, Montesi & Peressotti, 2019 & 2021

two-year process (with breaks: mostly lockdown entertainment)

- all results in the main theory were correct
- all proof strategies were correct
- formalization spotted unnecessary hypotheses and some typos
- difficulties suggested simplifications to the theory

Coq formalization

Cruz-Filipe, Montesi & Peressotti, 2019 & 2021

two-year process (with breaks: mostly lockdown entertainment)

- all results in the main theory were correct
- all proof strategies were correct
- formalization spotted unnecessary hypotheses and some typos
- difficulties suggested simplifications to the theory

the irony: two wrong results

- not in the main theory (separate sections)
- never questioned by reviewers

the lesson

formalization pays off

- the theorem prover is easier to convince than the reviewers
- it also requires less time
- once the formal proof is there, it must be right (right?)

the lesson

formalization pays off

- the theorem prover is easier to convince than the reviewers
- it also requires less time
- once the formal proof is there, it must be right (right?)

new experiment

submit an article claiming “all results have been formalized”

- (we’re anxiously awaiting the answer)

outline

- computer-assisted proofs are here to stay
- variety of flavors, important to understand them
- formalizations can actually help improve the theory
- still a lot of communication needed
- still hard for non-experts (but getting better)