Proofs and Things[1]

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

May 27, 2019

[1] Supported by the US National Science Foundation and SRI International.

Michael J. C. Gordon, FRS [1948–2017]

[a]

[a] Photo by Katriel Cohn-Gordon

Formalization, For and Against

Vannevar Bush in *As We May Think*, 1947: *Logic can become enormously difficult, and it would undoubtedly be well to produce more assurance in its use. . . . We may some day click off arguments on a machine with the same assurance that we now enter sales on a cash register.*

Alan Robinson: *We are still, alas, a very long way from knowing how informal proofs work. . . .*

If there comes a time when Woody's dream is fulfilled, one of the ways we shall know it will be not only that our machines will explain elegant informal proofs to us, but also that we will explain elegant informal proofs to our machines, and be confident that they can understand and appreciate what we are telling them.

de Millo, Lipton, and Perlis: *If the mathematical process were really one of strict, logical progression, we would still be counting on our fingers.*

Proofs and Things: Overview

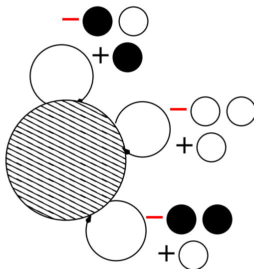
- The goal of the Big Proof program is to define the future direction of proof technology so that it is used by
  - Mathematicians to discover and verify new results
  - Scientists and engineers to apply mathematical modelling rigorously, and
  - Educators to effectively teach proofs and problems solving techniques
- Mathematics is the study of abstractions such as collections, maps, graphs, algebras, sequences, etc.
- It has been "unreasonably effective" as a language and a foundation for a growing number of other disciplines.
- In addition to education and pure mathematics pursuits, the growing need for scale and rigor in these other fields can be a fruitful source of challenges for Big Proof technology.
- The talk demonstrates a few simple PVS formalizations to illustrate the role of automation in creating coherent formalizations of mathematical content.

Finding Abstractions

Given a bag containing some black balls and white balls, and a stash of black/white balls. Repeatedly

1. Remove a random pair of balls from the bag
2. If they are the same color, insert a white ball into the bag
3. If they are of different colors, insert a black ball into the bag

What is the color of the last ball?



Proofs and Pigeons

- Can you allocate n cubbies to each of $n+1$ pigeons so that each pigeon gets its own cubby?.
- Let $m..n$ represent the subrange of integers from m to, but not including, n.
- The Pigeonole principle can be restated as asserting that there is no injection from $0..n+1$ to $0..n$.
- The Infinite Pigeonhole principle states that any finite partition of an infinite set must contain an infinite partition.
- Theorems of this sort are used routinely in computing.

Gilbreath's Card Trick

- Start with a deck consisting of a stack of quartets, where the cards in each quartet appear in suit order $\spadesuit, \heartsuit, \clubsuit, \diamondsuit$:

$$\begin{array}{l}\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle,\\ \langle K\spadesuit\rangle, \langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle,\\ \langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle\end{array}$$

- Cut the deck, say as $\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle, \langle K\spadesuit\rangle$ and $\langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle, \langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle$.
- Reverse one of the decks as $\langle K\spadesuit\rangle, \langle 8\diamondsuit\rangle, \langle Q\clubsuit\rangle, \langle 3\heartsuit\rangle, \langle 5\spadesuit\rangle$.
- Now shuffling, for example, as

$$\begin{array}{l}\langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \underline{\langle K\spadesuit\rangle}, \underline{\langle 8\diamondsuit\rangle},\\ \langle 4\diamondsuit\rangle, \langle 8\spadesuit\rangle, \underline{\langle Q\clubsuit\rangle}, \langle J\heartsuit\rangle,\\ \underline{\langle 3\heartsuit\rangle}, \langle 9\clubsuit\rangle, \underline{\langle 5\spadesuit\rangle}, \langle A\diamondsuit\rangle\end{array}$$

- *Each quartet contains a card from each suit. Why?*[2]

[2] Tony Hoare and Natarajan Shankar, *Unraveling a Card Trick*, 2010

Computing Majority

- Applications from Blockchain to elections need to compute majorities.
- An election has five candidates: Alice, Bob, Cathy, Don, and Ella.
- The votes have come in as:
  E, D, C, B, C, C, A, C, E, C, A, C, C.
- You are told that some candidate has won the majority (over half) of the votes.
- You successively remove pairs of dissimilar votes, until there are no more such pairs.
- Then the remaining votes, if any, are all for the same candidate, and that candidate is the winner.[3]

---

[3] R. S. Boyer and J. S. Moore, *MJRTY–A Fast Majority Vote Algorithm*, 1994.

Brief PVS Background

- PVS combines an expressive language close to mathematical vernacular with automation (SMT, rewriting, simplification) directed at efficient error detection/diagnosis in definitions, theorems, and proofs.
- PVS types are built from
  - Booleans, numbers, and type constants and parameters,
  - Using (dependent) tuple, record, and function types, algebraic datatypes/co-datatypes, and predicate subtypes.
- Expressions are built from
  - Variables and constants,
  - Using application, lambda abstraction, tuple/record construction/projection, (structural subtype polymorphic) updates, conditional expressions (`IF` and `CASES`, and let-expressions.

Higher-order Logic

- Types:
  - `bool` and `real` are types
  - $[T_1 \rightarrow T_2]$ and $[T_1, \ldots, T_n]$ are types if the T_i are.
- Products (in *n*-ary form) are useful so that functions don't have to be Curried.
- Terms:
  - Constants: `TRUE`, `FALSE`, 0, 1.
  - Variables
  - Application: f a
  - Abstraction: $\lambda(x : T) : t$
  - Pairing: (t_1, t_2)
  - Projections: $\texttt{PROJ}_i\ t$
- Polymorphic equality $b = c$ and conditional $\texttt{IF}[T](a, b, c)$, where a is of type `bool` and b and c are of type

(Dependent) Subtyping

- PVS extends higher-order logic with subtypes (e.g., primes, order/continuity-preserving operators) and dependent types (e.g., finite sequences)
  - Proof obligations are generated to check that an expression has the expected type.
  - Type system looks and feels like a mathematical vernacular, and catches lots of errors and missing arguments, e.g., $\binom{n}{k}$ is a positive integer.
  - Well-typed mathematics is coherent, and well-typed programs don't go wrong.

```
n: VAR nat
 factorial(n): RECURSIVE posint =
 (IF n = 0 THEN 1 ELSE n * factorial(n-1) ENDIF)
 MEASURE n
 n_choose_k(n, (k : upto(n))): posnat =
  factorial(n) / (factorial(k) * factorial(n - k))
```

Proofs and Fixpoints

```
Tarski_Knaster   [T : TYPE, ⊑ : PRED[[T, T]], ⊓ : [set[T] -> T] ]
                : THEORY
  BEGIN
   ASSUMING
    x, y, z: VAR T

    X, Y, Z : VAR set[T]  %synonym for [T -> bool]

    f, g : VAR [T -> T]

    reflexivity: ASSUMPTION  x ⊑ x

    antisymmetry: ASSUMPTION  x ⊑ y AND y ⊑ x IMPLIES x = y

    transitivity : ASSUMPTION x ⊑ y AND y ⊑ z IMPLIES x ⊑ z

    glb_is_lb: ASSUMPTION  X(x) IMPLIES ⊓(X) ⊑ x

    glb_is_glb: ASSUMPTION
       (FORALL x: X(x) IMPLIES y ⊑ x)
      IMPLIES y ⊑ ⊓(X)
   ENDASSUMING
```

Proofs and Fixpoints

```
  ⋮
  mono?(f): bool = (FORALL x, y: x ⊏ y IMPLIES f(x) ⊏ f(y))

  lfp(f) : T = ⊓(x | f(x) ⊏ x)

  fixpoint?(f)(x): bool =
    (f(x) = x)

  TK1: THEOREM
   mono?(f) IMPLIES
     lfp(f) = f(lfp(f))

END Tarski_Knaster
```

- Monotone operators on complete lattices have fixed points.
- Needs a nine-step proof: PVS proof steps follow those of a cogent textbook proof at a rate of .3 to 3 interactions per informal step.

Proofs and Automata

CS students are first exposed to proofs in automata theory: a deterministic finite automaton over an alphabet Σ, a state type, a start state, a transition function δ, and a set of final states.

```
DFA  [Sigma : TYPE,
       state : TYPE,
       start : state,
       delta : [Sigma -> [state -> state]],
       final? : set[state] ]
: THEORY
  BEGIN
   DELTA((string : list[Sigma]))((S : state)):
              RECURSIVE state =
     (CASES string OF
         null : S,
         cons(a, x): delta(a)(DELTA(x)(S))
      ENDCASES)
     MEASURE length(string)

   DAccept?((string : list[Sigma])) : bool =
       final?(DELTA(string)(start))
  END DFA
```

Nondeterministic Automata

Nondeterministic automata are defined in terms of a next-states function.

```
NFA   [Sigma : TYPE,
       state : TYPE,
       start : state,
       ndelta : [Sigma -> [state -> set[state]]],
       final? : set[state] ]
: THEORY
  BEGIN
    NDELTA((string : list[Sigma]))((s : state)) :
           RECURSIVE set[state] =
       (CASES string OF
         null : singleton(s),
         cons(a, x): lub(image(ndelta(a), NDELTA(x)(s)))
        ENDCASES)
     MEASURE length(string)

   Accept?((string : list[Sigma])) : bool =
     (EXISTS (r : (final?)) :
       member(r, NDELTA(string)(start)))
  END NFA
```

DFA/NFA Equivalence

The subset construction demonstrates that any NFA can be represented by a DFA, by interpreting the DFA types/operations using the NFA ones.

```
equiv[Sigma : TYPE,
       state : TYPE,
       start : state,
       ndelta : [Sigma -> [state -> set[state]]],
       final? : set[state] ]: THEORY
  BEGIN
    IMPORTING NFA[Sigma, state, start, ndelta, final?]
    dstate: TYPE = set[state]

    delta((symbol : Sigma))((S : dstate)): dstate =
        lub(image(ndelta(symbol), S))

    dfinal?((S : dstate)) : bool =
     (EXISTS (r : (final?)) : member(r, S))

    dstart : dstate = singleton(start)
   .
   .
   .
```

DFA/NFA Equivalence

Proofs are driven by commands like `grind` and `induct-and-simplify` which employ SAT/SMT solvers for simplification and failure diagnostics rather than as hammers.

```
    IMPORTING DFA[Sigma, dstate, dstart, delta, dfinal?]

    main: LEMMA
     (FORALL (x : list[Sigma]), (s : state):
        NDELTA(x)(s) = DELTA(x)(singleton(s)))

    equiv: THEOREM
     (FORALL (string : list[Sigma]):
     Accept?(string) IFF DAccept?(string))

   END equiv
```

Proofs and Programs

- Having seen automata, we can also represent programming calculi which are also ubiquitous in the theory of computing.
- *Proof-checking Metamathematics* (**PM**, 1986) formalizes both pure Lisp and lambda calculus (the Church–Rosser theorem).
- Many programming language theory proofs, e.g., POPLMark, and compiler correctness, e.g., Piton, micro-Gypsy, CompCert, CakeML, are examples of formalized metatheory over program calculi.

Proofs and Programs: Hoare Logic

- Hoare Logic metatheory was first formalized by Mike Gordon in HOL, and also by Gerwin Klein and Tobias Nipkow in Isabelle for their book *Concrete Semantics*.
- A Hoare triple has the form $\{P\}S\{Q\}$, where S is a program statement in terms of the program variables drawn from the set Y and P and Q are assertions containing logical variables from X and program variables.
- A program statement is one of
  1. A *skip* statement *skip*.
  2. A *simultaneous assignment* $\overline{y} := \overline{e}$ where $\overline{y}$ is a sequence of n distinct program variables, e is a sequence of n $\Sigma[Y]$-terms.
  3. A *conditional* statement $e\ ?\ S_1\ :\ S_2$, where C is a $\Sigma[Y]$-formula.
  4. A *loop while* e *do* S.
  5. A sequential composition $S_1; S_2$.

Let P, Q, C be state predicates.

Skip	$\{P\}skip\{P\}$
Assignment	$\{P[\overline{e}/\overline{y}]\}\overline{y} := \overline{e}\{P\}$
Conditional	$\dfrac{\{C \wedge P\}S_1\{Q\} \quad \{\neg C \wedge P\}S_2\{Q\}}{\{P\}C\ ?\ S_1\ :\ S_2\{Q\}}$
Loop	$\dfrac{\{P \wedge C\}S\{P\}}{\{P\}while\ C\ do\ S\{P \wedge \neg C\}}$
Composition	$\dfrac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$
Consequence	$\dfrac{P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$

Semantics: A trace σ of length $n > 1$ satisfies a triple $\{P\}S\{Q\}$ iff whenever $P(\sigma_0)$ and $\sigma \models S$, then $Q(\sigma_n)$.

Hoare Logic Semantics

- Both assertions and statements contain operations from a first-order signature Σ.
- An assignment σ maps program variables in Y to values in $dom(M)$.
- A program expression e has value $M[\![e]\!]\sigma$.
- The meaning of a statement $M[\![S]\!]$ is given by a sequence σ of states (of length at least 2).
- Hoare logic is sound if every provable triple is valid.
- The logic is complete if every if every valid triple has a proof.
- Soundness, completeness, and verification condition for the basic Hoare logic above is a simple student-level exercise in PVS thanks to the use of dependent subtypes and automation.

SRI International

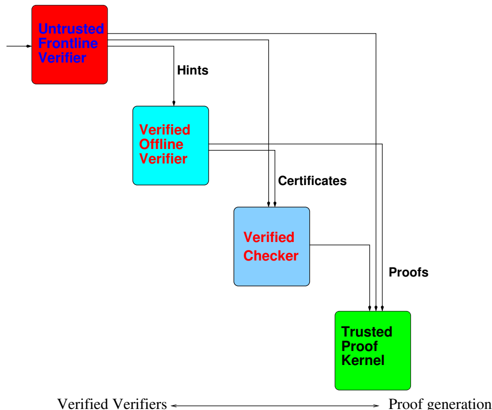
Proofs and "Proofs"

- As with Hoare Logic, proofs themselves can be represented in logic in order to do proofs about proofs.
- The tautology theorem from **PM** (and *Towards Mechanical Metamathematics*, JAR 1985) is an early example demonstrating that every propositional tautology has a proof.
- The incompleteness theorems are also about proofs:
  1. Any sufficiently expressive formalism Z is either inconsistent (proves A and $\neg A$) or incomplete (proves neither A nor $\neg A$, for some A).
  2. $Con(Z)$ encoded in Z is itself such an A.

Proofs and “Proofs”

- In 2016, a SAT solver was used to solve the Boolean Pythagorean Triples problem: Is it possible to 2-color that the positive integers such that there is no monochromatic Pythagorean triple, i.e., $\langle a, b, c \rangle$ where $a^2 + b^2 = c^2$?
- Up to $N = 7824$, it is possible to ensure that all Pythagorean triple are bichromatic, but not at $N = 7825$ — 200 terabyte proof which has been formally checked.
- The Schur number problem: What is the largest n such that there is a 5-coloring of the first n natural numbers without a monochromatic triple $a + b = c$? The answer is 160, and the SAT proof is two petabytes.

Proofs and Provers: Kernel of Truth (KoT)

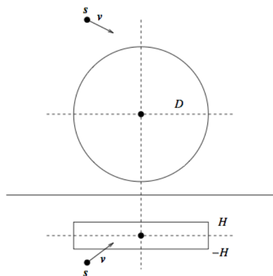


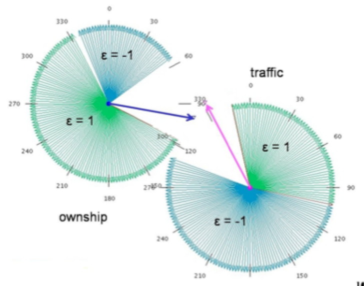
- A simple kernel checker for FOL(ZFC) proof certificates in PVS
- Verify other checkers relative to kernel or other verified checkers
- Instrument untrusted tools to generate certificates for verified checkers.

This works even if the verifier is untrusted since any circularity can be broken.

The verification of the verifier can be *certified* by expanding the proof to the kernel or to *previously* certified verifiers.

Proofs and Planes: Air-Traffic Control





- Extensive libraries, algorithms, meta-algorithms, policies, and standards (RTCA DO-365) comprehensively verified in PVS by NASA researchers and their collaborators.
- Related verification efforts covers geofencing for urban air space, floating point precision analysis, decision procedures for nonlinear arithmetic constraints.

Proofs and Inference Algorithms

- SAT/SMT solvers and theorem provers apply local inference steps to completion.
- An *Σ-inference structure* $\langle \Psi, \vdash, \Lambda, \mathcal{M} \rangle$ consists of
  - Ψ, a set of *logical states*
  - $\vdash$, the *reduction relation* between states
  - Λ, a map from states to Σ-formulas
  - $\mathcal{M}$, which extracts models from canonical states
- An *inference system* is an inference structure that is
  - *Conservative*: If $\psi \vdash \psi'$, then $\Lambda(\psi)$ and $\Lambda(\psi')$ are equisatisfiable.
  - *Progressive*: $\vdash$ is well-founded.
  - *Canonizing*: If $\psi \not\vdash \psi'$ for any ψ', then either ψ is $\bot$ (i.e., unsatisfiable) or ψ is in a canonical form so that $\mathcal{M}(\psi)$ is a model for $\Lambda(\psi)$.
- It is *strongly conservative* if whenever $\psi \vdash \psi'$, then ψ and ψ' are equisatisfiable and any model of ψ' is also a model of ψ.

What is an Inference Algorithm?

- An *inference algorithm* is an inference system where the reduction relation is presented as a collection of effective *inference rules* that transform an inference state ψ to an inference state ψ' such that $\psi \vdash \psi'$. **Example:** Ordered resolution is an algorithm for CNF satisfiability.
- Input K is a set of ordered clauses where the literals appear in decreasing order w.r.t. some order e.g., $q \prec \neg q \prec p \prec \neg p$.
- Tautologies, i.e., clauses containing both p and $\neg p$, are deleted from initial input.

Res	$\dfrac{K, p \vee \Gamma_1, \neg p \vee \Gamma_2}{K, p \vee \Gamma_1, \neg p \vee \Gamma_2, \Gamma_1 \vee \Gamma_2}$ $\Gamma_1 \vee \Gamma_2 \notin K$, $\Gamma_1 \vee \Gamma_2$ is not tautological
Contrad	$\dfrac{K, p, \neg p}{\bot}$

- A set of clauses is canonical if it is closed under applications of **Res** and the **Contrad** rule is inapplicable.

Resolution: Example

$$\cfrac{\cfrac{\cfrac{\cfrac{(K_0 =)\ \neg p \vee \neg q \vee r, \quad \neg p \vee q, \quad p \vee r, \quad \neg r}{(K_1 =)\ \neg q \vee r, \quad K_0}\ \textbf{Res}}{(K_2 =)\ q \vee r, \quad K_1}\ \textbf{Res}}{(K_3 =)\ r, \quad K_2}\ \textbf{Res}}{\bot}\ \textbf{Contrad}$$

- Drop the clause $\neg r$, and we reach an irreducible state from which a truth assignment $\{r \mapsto \top, q \mapsto \bot, p \mapsto \bot\}$ can be constructed.

Resolution as an Inference Algorithm

- The resolution inference system is *strongly conservative*: $\Gamma_1 \vee \Gamma_2$ is satisfiable if $p \vee \Gamma_1$ and $\neg p \vee \Gamma_2$ are.
- It is *progressive*: Bounded number of new clauses in the input variables.
- It is *canonizing*: Build a model M by assigning to atoms p_1 to p_n within a series of partial assignments $M_0, \ldots, M_n$:
  - M_0 is the empty truth assignment.
  - $M_{i+1} = M_i[p_{i+1} \mapsto v]$, where $v = \top$ iff there is some clause $p_{i+1} \vee \Gamma$ in the irreducible state K such that $M_i \models \neg\Gamma$.
- If $M_i \models \neg\Gamma$, then for any clause $\neg p_i \vee \Delta$, $M_i \models \Delta$ since $\Gamma \vee \Delta \in K$.
- **Invariant:** $M_i \models \Gamma$ for all clauses Γ in K in the atoms $p_1, \ldots, p_i$.
- Unordered resolution is also conservative, progressive, and canonizing, but it does not have the same set of canonical states.

Proofs and Computation: Dijkstra

- All computation is inference.
- Given a weighted directed graph $G = (V, W)$, with non-negative (or ∞) edge weights, find the smallest-weight path from a given source vertex s to each vertex, i.e., a map P_s on V: $P_s(s) = 0$, and for $v \neq s$,

$$P_s(v) = \bigsqcap\{P_s(u) + W(u, v) \mid u \in V\}.$$

- Let

$$post(X)(v) = \begin{cases} 0, \text{ if } v = s \\ \bigsqcap\{X(u) + W(u, v) \mid u \in dom(X)\}, otherwise. \end{cases}$$

- We therefore want to compute P_s such that $P_s = post(P_s)$.

Generalizing Inference Algorithms: Dijkstra

- The logical state has two partial maps D and Q:
  1. Each $v \in V$ is either in $dom(D)$ or $dom(Q)$, but not both,
  2. $D(v) = post(D)(v)$ for $v \in dom(D)$,
  3. $Q(v) = post(D)(v)$ for $v \in dom(Q)$, and
  4. $D(u) \leq Q(v)$ for $u \in dom(D)$ and $v \in dom(Q)$.
- Initially, $D = [s \mapsto 0]$, and $Q = [v \mapsto W(s, v) \mid v \neq s]$.
- Each inference step has the form

$$\frac{\langle D, Q\rangle}{\langle D', Q'\rangle}, \text{ where}$$

$$\begin{array}{rcl} u & = & argmin_v Q(v) \\ D' & = & D[u \mapsto Q(u)] \\ Q' & = & [v \mapsto Q(v) \sqcap (Q(u) + W(u, v)) \mid v \in dom(Q) - \{u\}] \end{array}$$

$$Algorithm = Inference + Strategy + Indexing.$$

Conclusions

- Mathematics has an ever-widening footprint going beyond the hard sciences to social sciences.
- Rigorous modeling and analysis requires formalization and mechanization for scale and accuracy.
- These application offer fertile ground for big proof technology that build on the formalization of core mathematical concepts and proofs.
- Automated reasoners can perform provably correct manipulations (e.g., algebra systems, SAT/SMT solvers), build models, generate formal proofs (theorem provers), check proofs (interactive proof assistants), and mine proofs (machine learning) for insights.
- Big proof technologies can help people (at every level of mathematical expertise) understand, create, and use mathematics effectively.