The Kernel of Truth[1]

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

Mar 3, 2010

[1]This research was supported NSF Grants CSR-EHCS(CPS)-0834810 and CNS-0917375.

Overview

- Deduction can be carried out by rigorous formal rules of inference.
- With mechanization, we can, in principle, achieve nearly absolute certainty, but in practice, there are many gaps.
- *How can we combine a high degree of automation in verification tools while retaining trust?*
- *Check the verification, but verify the checker.*
- *The Kernel of Truth* contains a network of verified checkers whose verifications have been checked relative (transitively) to a kernel checker.

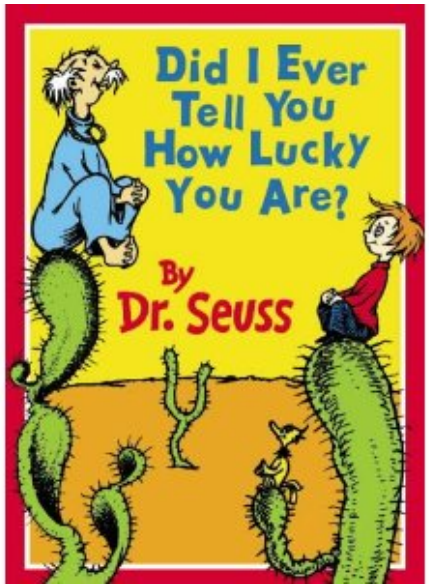
Robin and Amir

N. G. de Bruijn on Trust

. . . we ask whether this guarantee would be weakened by leaving the mechanical verification to a machine. This is a very reasonable, relevant and important question. It is related to proving the correctness of fairly extensive computer programs, and checking the interpretation of the specifications of those programs. And there is more: the hardware, the operating system have to be inspected thoroughly, as well as the syntax, the semantics and the compiler of the programming language. And even if all this would be covered to satisfaction, there is the fear that a computer might make errors without indicating them by total breakdown.

I do not see how we ever can get to an absolute guarantee. But one has to admit that compared to human mechanical verification, computers are superior in every respect.

Did I Ever Tell You How Lucky You are? [Dr. Seuss]



Oh, the jobs people work at!
Out west, near Hawtch-Hawtch,
there's a Hawtch-Hawtcher Bee-Watcher.
His job is to watch . . .
is to keep both his eyes on the lazy town bee.
A bee that is watched will work harder, you see.
Well . . . he watched and he watched.
But, in spite of his watch,
that bee didn't work any harder. Not mawtch.
So then somebody said,
"Our old bee-watching man
just isn't bee-watching as hard as he can.
He ought to be watched by another Hawtch-Hawtcher.
The thing that we need
is a Bee-Watcher-Watcher."
WELL . . .
The Bee-Watcher Watcher watched the Bee-Watcher.
He didnt watch well. So another Hawtch-Hawtcher
had to come in as a Watch-Watcher-Watcher.
And today all the Hawtchers who live in Hawtch-Hawtch
are watching on Watch-Watcher-Watchering-Watch,
Watch-Watching the Watcher who's watching that bee.
You're not a Hawtch-Hawtcher. You're lucky you see.

Trusting Inference Procedures

- Absolute proofs of consistency are ruled out by Gödel's second incompleteness theorem, but relative consistency proofs can be quite useful.
- We could hope for correctness relative to a kernel proof system, as in the foundational systems Automath and LCF.
- *Caveat: LCF-based systems have been known to have unsound kernels.*

Proof Generation to Verified Inference Procedures

- If we accept only those claims that have valid formal proofs, then we have a spectrum of options.
- At one extreme, we can *generate* formal proofs that are validated by a primitive proof checker.
- This kernel proof checker and its runtime environment will have to be trusted.
- *Proof generation imposes a serious time, space, effort overhead.*
- At the other extreme, we can *verify* the inference procedure by proving that every claim has a proof.
- We have to trust the inference procedures used in this verification.

Verifying the Verifier Reflexively

- Reflection was first introduced in the seventies with Davis/Schwarz, Weyhrauch, and Boyer/Moore's metafunctions.
- The syntax of the logic, or a fragment of the logic, is encoded in the logic itself and the tactics are essentially proved correct.
- In *computational* reflection, we define an *interpreter* for the reflected syntax of a fragment of the logic, e.g., arithmetic expressions, and construct a *verified simplifier*.
- Computational reflection (metafunctions) can be directly implemented in any logic that supports syntactic representation and evaluation.
- Chaieb and Nipkow show that the reflected quantifier elimination procedures runs 60 to 130 times faster than the corresponding tactic.

Proof Reflection

- In proof reflection, we represent formal proofs and show that a new inference rule is derivable.
- For example, we can define a predicate $\mathit{Provable}(A)$ and establish that $\mathit{Provable}(f(A)) \implies \mathit{Provable}(A)$.
- Jared Davis has built a fairly sophisticated self-verified prover *Milawa*, incorporating induction, rewriting, and simplification.
- He defines 11 layers of proof checkers of increasing sophistication so that proofs at level $i+1$ can be justified by proofs at level i, for $1 \leq i \leq 10$.
- J. Moore has a talk on Milawa on Thursday.

Verifying Inference Procedures (Non-reflectively)

- Instead of reflection, one can just use a verification system to verify decision procedures.
- There is a long history of work in verifying decision procedures, including
  1. Satisfiability solvers
  2. Union-Find
  3. Shostak combination
  4. BDD packages
  5. Gröbner basis computation
  6. Presburger arithmetic procedures
  7. Explicit-state model checker (Besc)
- However, these procedures are not comparable in performance to state-of-the-art implementations.

Should Verifiers be Verified?

- Short answer: *NO!*
- *There's many a slip betwixt cup and lip* with respect to software. Verifying the verifier will only marginally impact software reliability or quality.
- Effective tools tend to be highly experimental in construction as well as in their usage.
- It would be hard for verification to keep up with the cutting edge in tool development.
- However, it does make sense for verifiers to generate certificates ranging from proofs to witnesses.
- These certificates can be checked offline by verified checkers.

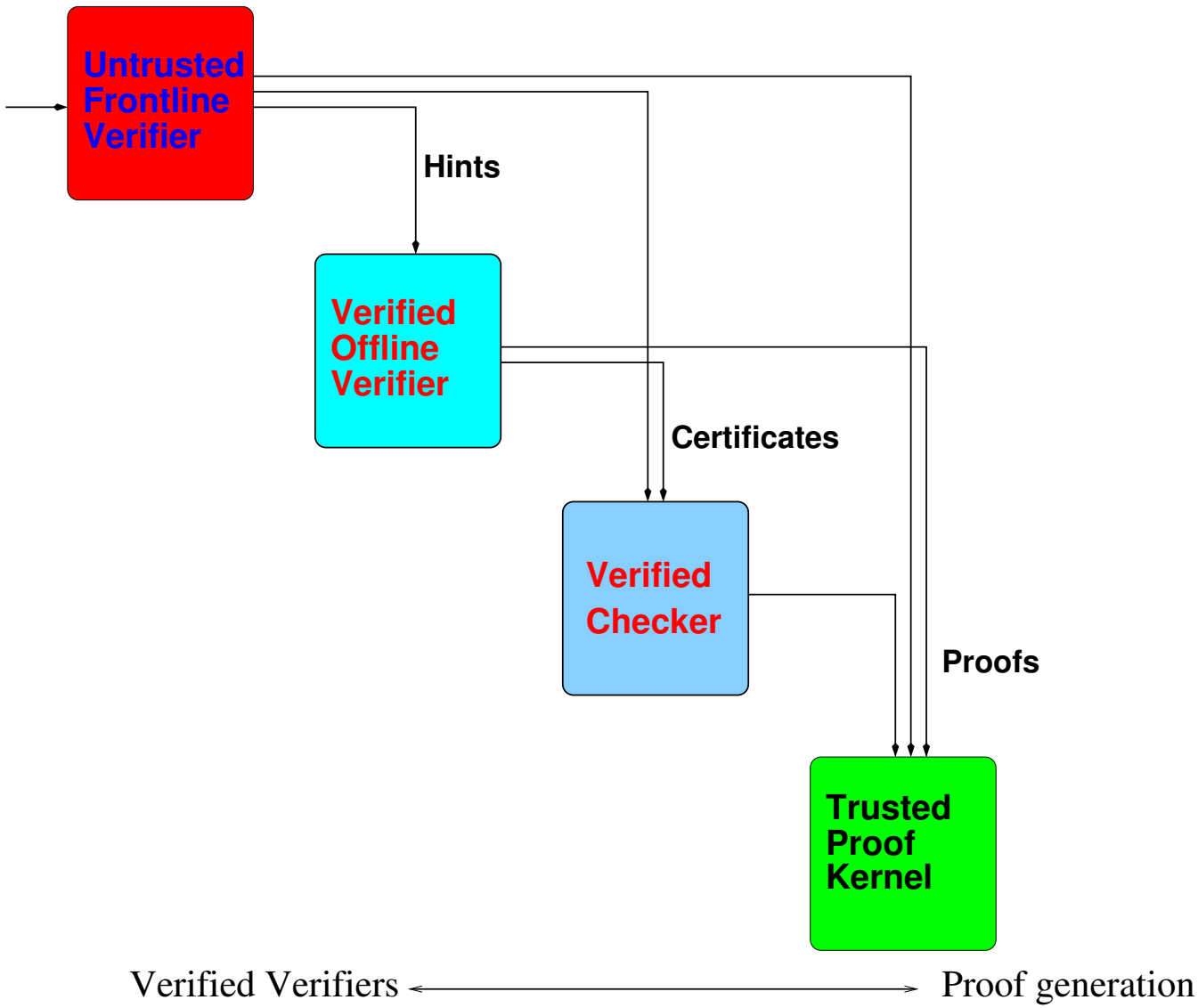
The PVS Language

- The PVS logic is based on higher-order logic.
- Predicate subtypes and dependent types can be used to capture even numbers, partial ordering relations, injective functions, finite sequences, and order-preserving maps as types.
- Theorem proving and type-checking are intertwined.
- Specifications are structured as theories which are lists of type, constant, and formula (assumptions, axioms, or theorems) declarations.
- Theories can be parametric in constants, types, and other theories, with theory interpretations.
- The PVS type checker is a very complex piece of software that does type inference and proof obligation generation.

PVS Inference Procedures

- Proofs in PVS are constructed within a classical sequent calculus.
- Proofs are developed by means of interactive proof commands.
- Each proof command can either invoke a defined strategy or a primitive proof step.
- Some of the internal primitive proof steps are quite complex; others invoke external tools like BDD packages, MONA, RAHD, and Yices.
- For example, the PVS simplifier uses a complex combination of decision procedures and rewriting to carry out arithmetic, Boolean, array, datatype, and other simplifications.
- Matching, rewriting, and simplification use decision procedures.
- *How can we trust the claims arising from such inference procedures?*

Kernel of Truth



The Kernel of Truth (KoT)

- The kernel contains a reference proof system formalizing ZFC.
- It also contains several verified checkers for specialized certificate formats.
- If the checker validates the certificate for a claim, then there is a proof of the claim.
- These certificates can be more compact than proofs.
- Generating and checking certificates is easier than generating proofs.
- Proof generation (including LCF) and verification are subsumed.
- Verifying the checkers is (a lot) easier than verifying the inference procedures.
- *But, why should we trust the latter verification?*

The Kernel Proof Checker: Syntax

- The kernel proof checker is built on first-order logic.
- The symbols consist of variables, function symbols, predicate symbols, and quantifiers.
- Function and predicate symbols can be interpreted or uninterpreted.
- Interpreted symbols are used for the defined operations.
- Uninterpreted symbols are used as schematic variables, e.g., Skolem constants.
- The basic propositional connectives are $\vee$ and $\neg$, and the existential quantifier $\exists$ is chosen as basic.

Kernel Proof Checker: One-Sided Sequents

Ax	$\dfrac{}{\vdash A, \neg A, \Delta}$
$\neg\neg$	$\dfrac{\vdash A, \Delta}{\vdash \neg\neg A, \Delta}$
$\vee$	$\dfrac{\vdash A, B, \Delta}{\vdash A \vee B, \Delta}$
$\neg\vee$	$\dfrac{\vdash \neg A, \Delta \quad \vdash \neg B, \Delta}{\vdash \neg(A \vee B), \Delta}$
Cut	$\dfrac{\vdash A, \Delta \quad \vdash \neg A, \Delta}{\vdash \Delta}$

The other connectives can be defined in terms of $\neg$ and $\vee$.

Kernel Proof Checker: Quantifiers

$\exists$	$\dfrac{\vdash A[t/x], \Delta}{\vdash \exists x.A, \Delta}$
$\neg\exists$	$\dfrac{\vdash \neg A[c/x], \Delta}{\vdash \neg\exists x.A, \Delta}$
f	$\dfrac{\vdash \Delta}{\vdash \Delta[\lambda\overline{x}.s/f]}$
p	$\dfrac{\vdash \Delta}{\vdash \Delta[\lambda\overline{x}.A/p]}$

The uninterpreted constant c in $\neg\exists$ must not occur in the conclusion, and there are no free variables in t, $\lambda\overline{x}.s$, $\lambda\overline{x}.A$.
The universal quantifier $\forall$ can be defined as a *macro* in terms of $\exists$.

Kernel Proof Checker: Equality

Equality is an interpreted predicate.

Reflex	$\dfrac{}{\vdash a = a, \Delta}$
Predicate Congruence	$\dfrac{\vdash a_1 = b_1, \Delta \quad \ldots \quad \vdash a_n = b_n, \Delta}{\vdash p(a_1, \ldots, a_n), \neg p(b_1, \ldots, b_n), \Delta}$

Transitivity, symmetry, and function congruence can be derived from reflexivity and predicate congruence.

Formalizing ZFC

- The axioms of ZFC are added to the core first-order logic.
- Uninterpreted predicates can be used for formulating axiom schemes as axioms.
- For example, the *comprehension* axiom scheme of set theory can be written as

$$\forall y.\exists z.\forall x.(x \in z \iff x \in y \wedge p(x)),$$

  where p is a schematic predicate.
- Here, p can be replaced by a lambda-expression of the form $\lambda w.A$ to yield $\forall y.\exists z.\forall x.x \in z \iff x \in y \wedge A[x/w]$.
- Similarly, the *replacement* axiom scheme can be written as

$$\forall w.\left(\begin{array}{ll} & (\forall x \in w.\exists! y.q(x,y,w)) \\ \Longrightarrow & \exists z.\forall y.(y \in z \iff \exists x \in w.q(x,y,w)) \end{array}\right)$$

  where q is a schematic predicate.

Verified Checkers: Resolution

- Resolution can be used to construct a proof of $\bot$ from a set of clauses K.
- More generally, resolution can be used to construct the proof of a clause κ from some subset of clauses in K.
- Each resolution step where a clause κ is derived from the clauses κ_1 and κ_2 is represented by the proof of the sequent $\vdash \neg\kappa_1, \neg\kappa_2, \kappa$.
- This proof can be constructed using Ax, $\vee$, and $\neg\vee$.

Verified Checkers: Logic Front-Ends

- We have seen how the KoT kernel can be used to verify checkers for inference procedures (e.g., rewriting) and proof formats (e.g., resolution).
- The kernel can also be used as the back-end for various logics, e.g., equational logic, higher-order logic and modal, temporal, and program logics.
- This is done by giving a ZFC semantics for these logics.
- The proof rules for the logic are then justified relative to this semantics.
- Note that the ZFC part of the kernel is not needed for certifying purely logical claims such as those generated using resolution or rewriting.

Trusting the Verified Checker

- *Since the checkers have been verified by untrusted tools, can we trust the checker?*
- The untrusted verifier U has its results checked by V.
- Suppose that V is also capable of generating a proof.
- And, we have used U to verify V.
- Then, we can also generate an independently checkable proof for the correctness of V as verified by V.
- So that there is no need to trust V with its own verification.

A Hierarchy of Checkers

- Many inference tools can have their claims certified relative to other inference tools.
- For example, the computations of a BDD package can be certified by a SAT solver.
- Similarly, a static analysis tool can be certified by an SMT solver.
- An SMT solver can itself be certified using a SAT solver and certificate checkers for the individual theories.
- A SAT solver can be certified by generating resolution proofs.
- But we can also have verified reference tools, like a verified SAT or theory solver.
- Claims that are reducible to a common foundation can be shared across different systems.

SAT as a Kernel Core

- Propositional satisfiability (SAT) is the problem of checking if a Boolean formula ϕ has a truth assignment M such that $M \models \phi$.
- In particular, if ϕ is unsatisfiable, then $\neg\phi$ is valid.
- The validation of many verifiers can be reduced to SAT plus a little bit.
- SAT can therefore be used as a key component of a kernel that can be used to check claims generated by other untrusted solvers.

Reduction to SAT: Binary Decision Diagrams

- BDD packages provide an operation `sum_of_cubes` to extract the disjunctive normal form.
- If BDD G_ϕ represent the formula ϕ, let $\sigma_1 \vee \ldots \vee \sigma_n$ be the sum-of-cubes representation of G_ϕ, with $\kappa_1 \wedge \ldots \wedge \kappa_n$ as the CNF of its negation.
- The correspondence between G_ϕ and ϕ can be checked by testing the satisfiability of $\phi \wedge \kappa_1 \wedge \ldots \wedge \kappa_n$ and $\neg\phi \wedge \sigma_i$, for $1 \leq 1 \leq n$.

Reduction to SAT: Symbolic Model Checking

- In symbolic model checking, we have a transition system *model* M given by $\langle I, N \rangle$ with an initial set of states I and a transition relation N.
- A formula ϕ holds in the model if $M \models \phi$.
- The set of reachable states is the smallest set of states containing I and closed under the image operation with respect to N.
- The set R is an *overapproximation* of the reachable states if $\neg I(s) \wedge R(s)$ and $R(s) \wedge N(s, s') \wedge \neg R(s')$ are both unsatisfiable.
- **AG**P holds if $R(s) \wedge \neg P(s)$ is unsatisfiable.
- **AF**P can be validated by a sequence of sets $S_0, \ldots, S_n$ such that $S_0(s) \wedge \neg P(s)$, $S_{i+1}(s) \wedge N(s, s') \wedge \neg S_i(s')$ for each $i \leq n$, and $I(s) \wedge \neg S_0(s) \wedge \ldots \wedge \neg S_n(s)$ are all unsatisfiable.

Reduction to SAT+: SMT

- A theory is a set of models closed under isomorphism.
- A formula ϕ is $\mathcal{T}$-satisfiable for theory $\mathcal{T}$ if there is an $M \in \mathcal{T}$ such that $M \models \phi$.
- An SMT solver checks the *theory satisfiability* of a formula.
- When ϕ is unsatisfiable, it generates *theory lemmas* θ, such that θ is $\mathcal{T}$-valid and $\theta \wedge \phi$ is propositionally unsatisfiable.
- The theory lemmas θ can be supported by proofs or by certificates that can be checked by a verified checker.

Certificates for Theory Lemmas

- For example, certificates for arithmetic proofs can be obtained from results like
  1. Farkas lemma: Either $Ax \leq b$ or $y^T A = 0, y^T b = -1$ is solvable, but not both.
  2. Hilbert's (weak) Nullstellensatz: If P is a set of polynomials, and I is the ideal generated by P, then $P = 0$ has no solutions iff $1 \in I$.
  3. Stengle's Positivstellensatz: Given polynomial sets P, Q, and R, the constraints $P \geq 0$, $Q = 0$, and $R \neq 0$ is unsolvable iff $p + q + (\Pi R)^{2n} = 0$ for some $p \in Cone(P)$, $g \in Ideal(Q)$, and $n \geq 0$.

Reduction to SAT+: Quantified Logic

- Herbrand's theorem asserts that if a formula in prenex form is unsatisfiable, then some finite conjunction of ground Herbrand instances is unsatisfiable.
- For example, the formula $\forall x.\exists y.P(x) \wedge \neg P(y)$ can be Herbrandized as $\forall x.P(x) \wedge \neg P(f(x))$.
- The ground Herbrand instance $(P(c) \wedge \neg P(f(c))) \wedge (P(f(c)) \wedge \neg P(f(f(c))))$.
- Herbrand's theorem for first-order logic (without equality) can be used to reduce the validation of first-order proofs to SAT.
- Similarly, Herbrand's theorem for first-order logic with equality and higher-order logic can be used to reduce these logics to SMT (SAT + EUF).

Conflict-Driven Clause Learning (CDCL) SAT

Name	Rule	Condition
Propagate	$\dfrac{h, \langle M \rangle, K, C}{h, \langle M, l[\Gamma] \rangle, K, C}$	$\Gamma \equiv l \vee \Gamma' \in K \cup C$ $M \models \neg\Gamma'$
Decide	$\dfrac{h, \langle M \rangle, K, C}{h+1, \langle M; l[] \rangle, K, C}$	$M \not\models l$ $M \not\models \neg l$
Conflict	$\dfrac{0, \langle M \rangle, K, C}{\bot}$	$M \models \neg\Gamma$ for some $\Gamma \in K \cup C$
Backjump	$\dfrac{h+1, \langle M \rangle, K, C}{h', \langle M_{\leq h'}, l[\Gamma'] \rangle, K, C \cup \{\Gamma'\}}$	$M \models \neg\Gamma$ for some $\Gamma \in K \cup C$ $\langle h', \Gamma' \rangle$ $= \mathit{analyze}(\psi)(\Gamma)$ for $\psi = h, \langle M \rangle, K, C$

CDCL Example

- Let K be
  $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$.
-

step	h	M	K	C	Γ
select s	1	$; s$	K	$\emptyset$	_
select r	2	$; s; r$	K	$\emptyset$	_
propagate	2	$; s; r, \neg q[\neg q \vee \neg r]$	K	$\emptyset$	_
propagate	2	$; s; r, \neg q, p[p \vee q]$	K	$\emptyset$	_
conflict	2	$; s; r, \neg q, p$	K	$\emptyset$	$\neg p \vee q$

CDCL Example (contd.)

step	h	M	K	C	Γ
conflict	2	$; s; r, \neg q, p$	K	$\emptyset$	$\neg p \vee q$
backjump	0	$\emptyset$	K	q	_
propagate	0	$q[q]$	K	q	_
propagate	0	$q, p[p \vee \neg q]$	K	q	_
propagate	0	$q, p, r[\neg p \vee r]$	K	q	_
conflict	0	q, p, r	K	q	$\neg q \vee \neg r$

With Marc Vaucher, we have verified a CDCL SAT solver.

CDCL Certificate

- We can build compact, easily checkable resolution certificates.

Num.	Clause	Proof
0	$p \vee q$	
1	$\neg p \vee q$	
2	$p \vee \neg q$	
3	$\neg p \vee r$	
4	$\neg q \vee \neg r$	
5	q	$0, 1$
6	p	$5, 2$
7	r	$3, 6$
8	$\bot$	$4, 5, 7$

- With Andrei Dan and Antoine Toubhans, we have defined and verified an executable trace checker for PicoSAT proof traces.

Certified Algorithms

- McConnell, Mehlhorn, Näher, and Schweitzer write in *Certifying Algorithms* (2010):

> *A user of a certifying algorithm inputs x and receives the output y and the witness w. He then checks that w proves that y is a correct output for input x. The process of checking w can be automated with a checker, which is an algorithm for verifying that w proves that y is a correct output for x. In may cases, the checker is so simple that a trusted implementation of it can be produced, perhaps even in a different language where the semantics are fully specified. A formal proof of correctness of the implementation of the certifying algorithm may be out of reach, however, a formal proof of the correctness of the checker may be feasible*

- The verification of checker routines is a fruitful application for formal methods.

Conclusions

- Inference tools, even simple ones, do have bugs.
- Sometimes these bugs can lead to unsoundness.
- Verifying working inference procedures can be a fruitless exercise.
- Proof generation imposes a high overhead, particularly for experimental tools.
- The Kernel of Truth approach: *Check the verification, but verify the checker.*
- This approach can be applied more generally to computing beyond verification.