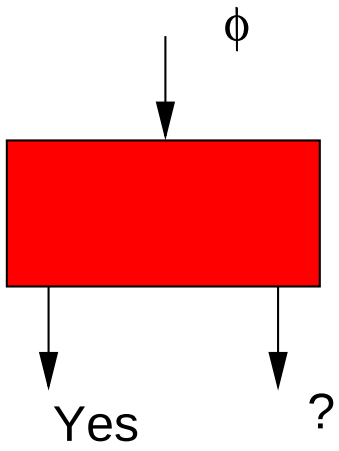
A "Self-Verifying" Theorem Prover

Jared Davis

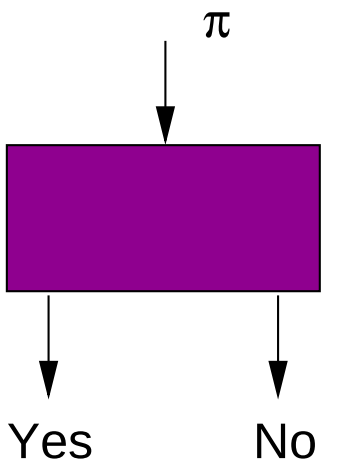
(advertisement by J Strother Moore)

Department of Computer Sciences
University of Texas at Austin

September 18, 2009



Theorem Prover



Proof Checker

Rules of Inference

Prop Schema $\dfrac{}{\neg A \vee A}$

Contraction $\dfrac{A \vee A}{A}$

Expansion $\dfrac{A}{B \vee A}$

Associativity $\dfrac{A \vee (B \vee C)}{(A \vee B) \vee C}$

Cut $\dfrac{A \vee B,\ \neg A \vee C}{B \vee C}$

Instantiation $\dfrac{A}{A/\sigma}$

Induction (ordinals below ϵ_0)

Rec Defn (ordinals below ϵ_0)

Axioms

Reflexivity

$$x = x$$

Equality

$$x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_1 = x_2 \rightarrow y_1 = y_2$$

Functional Reflexivity

$$x_1 = y_1 \rightarrow \ldots \rightarrow x_n = y_n$$

$$\rightarrow$$

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Beta Reduction

$$((\lambda x_1 \ldots x_n \ .\beta)\ t_1, \ldots, t_n)$$

$$=$$

$$\beta/[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$$

Base Evaluation

e.g., $1 + 2 = 3$

52 Lisp Axioms

e.g., $car(cons(x, y)) = x$

Assumed Characteristics

Proof Checker: Small (1500 LOC), Trusted, Impractical

Theorem Prover: Big (100K LOC), Untrusted, Practical

How can we trust the Theorem Prover?

Related Work

LCF-style (trust depends on type system, time-inefficient)

Constructive type theory (trust depends on type system, space-inefficient)

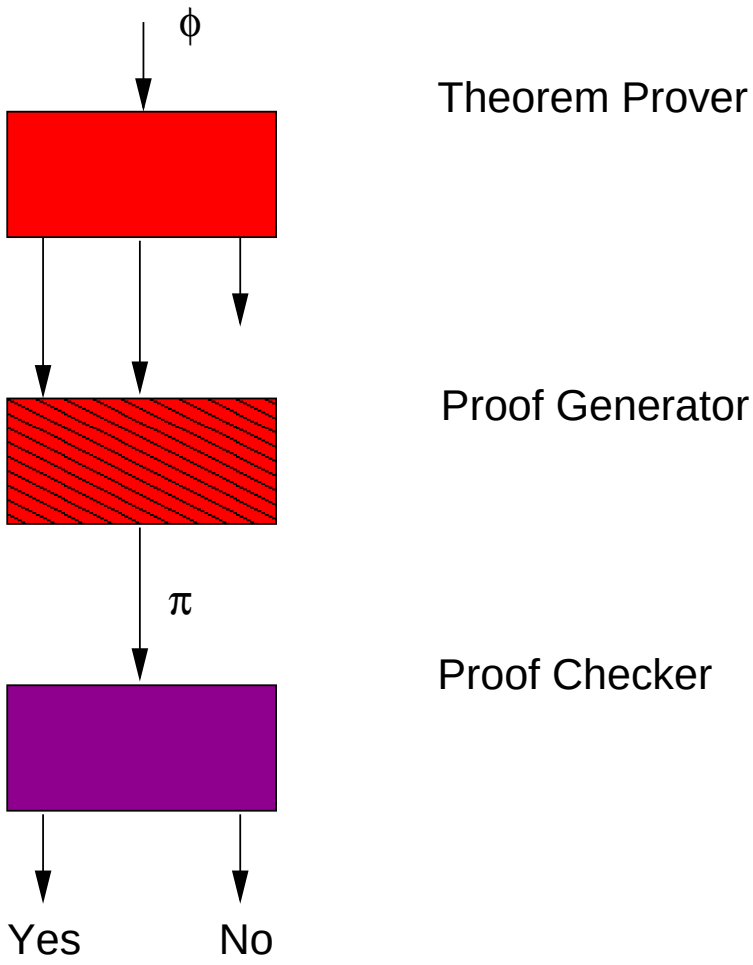
Proof Objects (trust depends on proof checker, space- and time-inefficient)

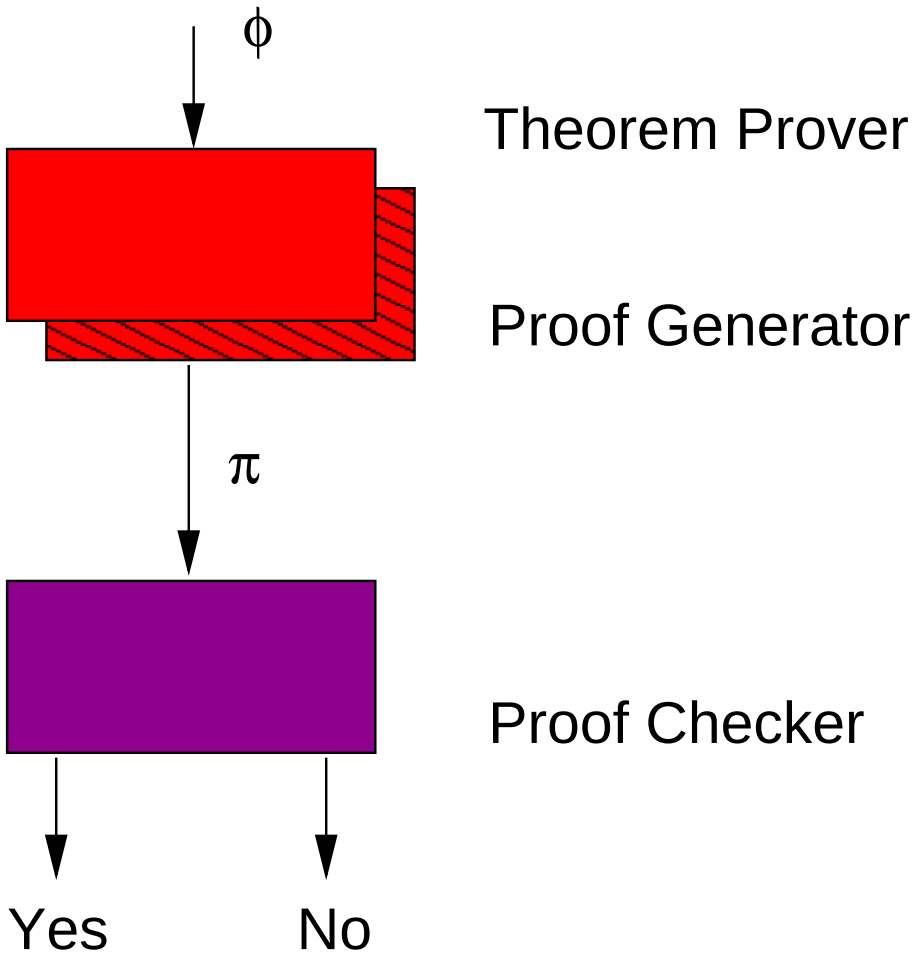
Related Work

LCF-style (trust depends on type system, time-inefficient)

Constructive type theory (trust depends on type system, space-inefficient)

Proof Objects (trust depends on proof checker, space- and time-inefficient)


ϕ
Theorem Prover
Proof Generator
π
Proof Checker
Yes
No


ϕ
Theorem Prover
Proof Generator
π
Proof Checker
Yes
No

Two Alternatives

(1) Run the Proof Generator every time and check the proof with the trusted Proof Checker.

(2) Prove that the Proof Generator will always generate a proof that succeeds.

Two Alternatives

(1) Run the Proof Generator every time and check the proof with the trusted Proof Checker.

(2) Prove that the Proof Generator will always generate a proof that succeeds.

Two Alternatives

(1) Run the Proof Generator every time and check the proof with the trusted Proof Checker.

(2) Prove that the Proof Generator will always generate a proof that succeeds.

But what prover do you use?

Correctness wrt Proof Checker ("Fidelity")

When Theorem Prover ("A") returns "Yes" on ϕ,

- Proof Generator produces a well-formed proof π
- Proof π concludes with ϕ
- Proof Checker ("C") accepts π

The Project

Suppose you've defined the proof checker C as an executable Lisp program. Then use it to

- admit the definition of C as an axiom
- admit the definition of A as an axiom
- check a proof of the correctness formula:

Correctness Formula

$$formula(\phi) \wedge A(\phi)$$

$$\longrightarrow$$

$$(\exists \pi.\ proof(\pi) \wedge concl(\pi) = \phi \wedge C(\pi))$$

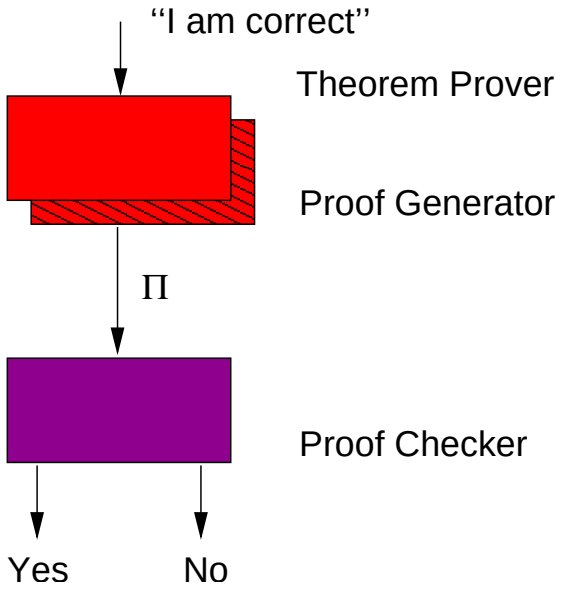
What You Must Trust

- the program C
- the hardware/software platform it runs on
- the statement of the correctness theorem (you needn't bother to read the definition of A if you don't care how it works)
- the fact that there is a proof file that C certifies as a proof of the statement

Jared's Problem

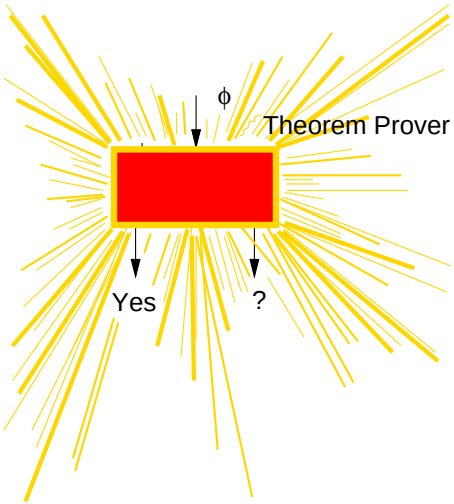
generating a checkable proof of the correctness statement

Plan



- Prove "I am correct" with Theorem Prover
- Generate *that* proof Π
- Check Π with Proof Checker
- Never generate another proof

Plan



- Prove "I am correct" with Theorem Prover
- Generate *that* proof Π
- Check Π with Proof Checker
- Never generate another proof

Unfortunately

The proof of correctness, Π, of a practical theorem prover is too big to generate and check.

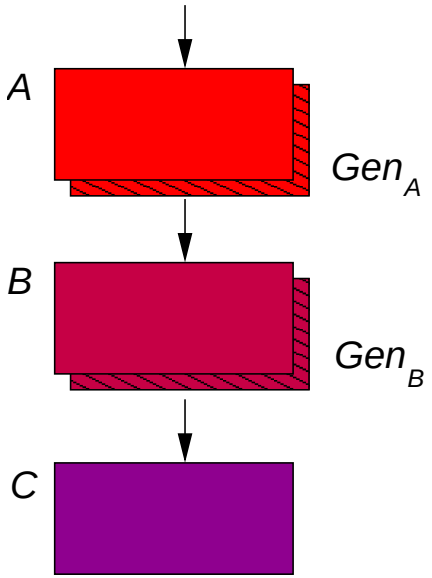
...because

- to be trustworthy, the Proof Checker takes *tiny* inference steps, so proofs are big, and
- the Theorem Prover is a *big* system

Solution (. . .**sort of**)

Introduce a more powerful trusted proof checker and prove it correct.

Solution (. . .sort of)



- Use A to prove A correct wrt B
- Run Gen_A to get B-Level proof Π_A
- Use A to prove B correct wrt C
- Run $Gen_B \circ Gen_A$ to get C-Level proof Π_B
- Check Π_B with C
- Check Π_A with B

Solution (. . .sort of)

Let $\Gamma = Gen_A(Gen_B(\Pi_A))$.

Then:

Γ is a C-level proof of the correctness of A

Γ is certified by C

Γ is (might be) too large to actually construct

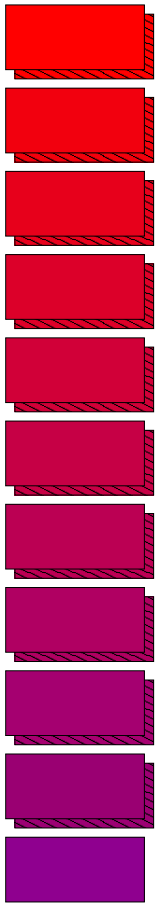
Unfortunately

Just one intermediate proof checker is not enough, i.e., even Π_A and Π_B are too large to construct.

It is important to

- increase the size of the inference step, *and*
- decrease the complexity differences between the software systems

Jared's Stack



Level

11 Induction and other tactics

10 Conditional rewriting

9 Evaluation and unconditional rewriting

8 Audit trails (in prep for rewriting)

7 Case splitting

6 Factoring, splitting help

5 Assumptions and clauses

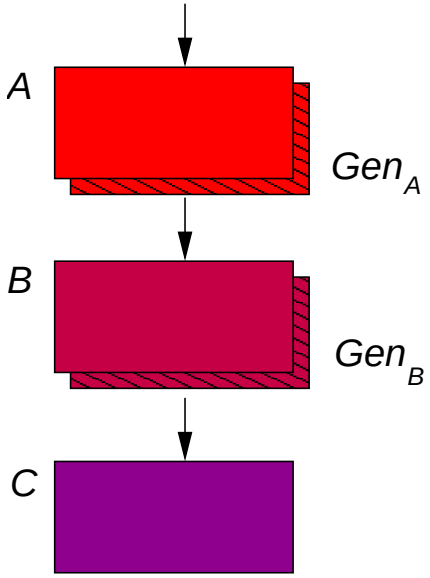
4 Miscellaneous ground work

3 Rules about primitive functions

2 Propositional reasoning

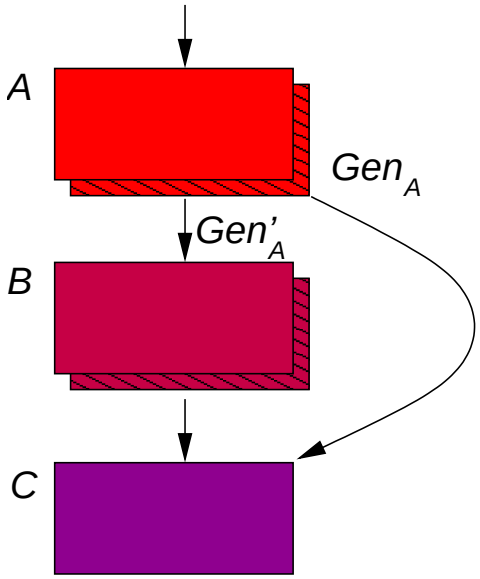
1 Primitive proof checker

Solution (. . .sort of)



- Use A to prove A correct wrt B
- Run Gen_A to get B-Level proof Π_A
- Use A to prove B correct wrt C
- Run $Gen_B \circ Gen_A$ to get C-Level proof Π_B
- Check Π_B with C
- Check Π_A with B

Solution (. . .more accurately)



- Use A to prove A correct wrt C
- Run Gen'_A to get B-Level proof Π_A
- Use A to prove B correct wrt C
- Run Gen_A to get C-Level proof Π_B
- Check Π_B with C
- Check Π_A with B

Why Do It This Way?

Because when Jared was exploring for the proof he did not know where the boundaries would be between the various intermediate proof checkers.

It was easier to always reason about the existence of a C-level proof so he didn't have to change the purported proof of A when he introduced a new feature in B.

Gen'_A is like Gen_A but uses B-level steps when possible.

Gen'_A is actually obtained from Gen_A by redefining subroutines that generate the explanations for certain steps.

Gen'_A need not be verified. If the one proof it generates, Π_A, checks out, you're done.

Proof Sizes (Gigabytes*)

Level	*Defs*	*Thms*	*Max Sz*	*Sum Sz*
1	201	2,015	2.8	51.4
2	87	514	2.7	72.3
3	230	815	4.9	63.9
4	168	991	9.2	152.9
5	192	1,071	3.7	74.6
6	55	402	6.0	26.2
7	83	749	3.5	7.5
8	184	1,059	5.6	54.4
9	427	2,475	1.5	12.3
10	82	616	1,934.3	2,713.9
11	233	1,157	0.2	21.4

* 1 `cons` = 8 bytes

Is Level 11 Practical?

It is good enough to prove the correctness of itself (100K LOC) and of all the lower levels.

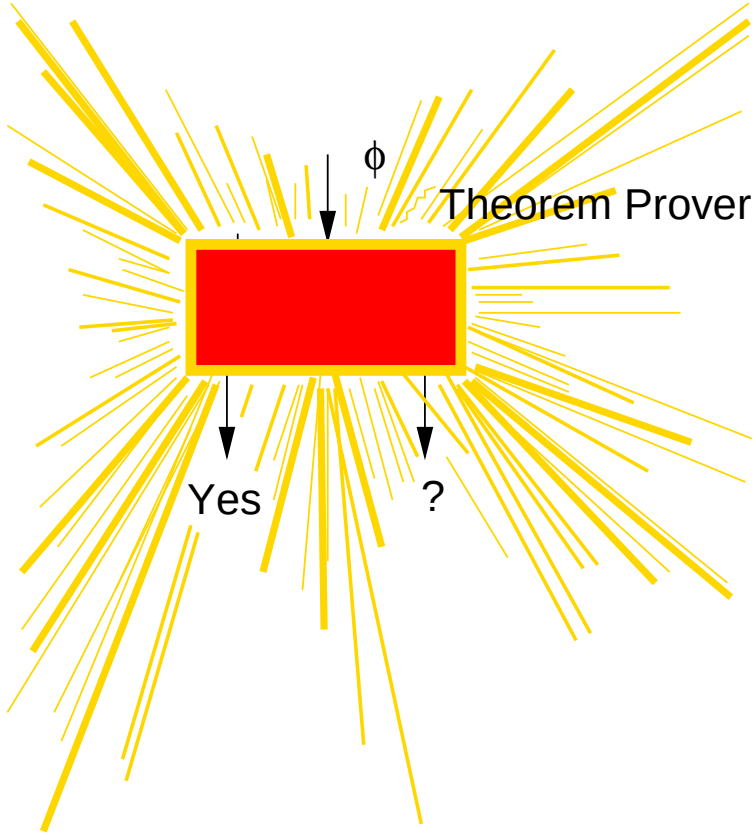
Reproducibility

To reduce the chances that implementation or hardware bugs invalidate his proofs, the proofs have been checked on 11 combinations of 4 machines (AMD and Intel processors), 3 Linux variants, and 4 Common Lisps (CCL, CMUCL, SBCL, and CLISP).

The fastest takes 19 hours to check all the proofs.

The slowest takes 13 days.

Conclusion



References

http://www.cs.utexas.edu/∼jared/milawa/Web/