Formal Verification of Mathematical Algorithms

John Harrison
Intel Corporation

Real Numbers and Computers 5

Lyon

5th September 2003

Bugs in computer systems

Most modern computer systems contain ‘bugs’. These can usually be coped with:

- Fixes in later versions
- Workarounds
- Documentation changes

But the consequences can be more spectacular or even deadly.

- Ariane
- Therac machine

Floating-point bugs

Even when not a matter of life and death, the financial consequences of a bug can be very serious:

- 1994 FDIV bug in the Intel®Pentium® processor: US $500 million.
- Today, new products are ramped much faster and a similar bug might be even more expensive.

So Intel is especially interested in all techniques to reduce errors.

Complexity of designs

At the same time, market pressures are leading to more and more complex designs where bugs are more likely.

- A 4-fold increase in bugs in Intel processor designs per generation.
- Approximately 8000 bugs introduced during design of the Pentium 4.

Fortunately, pre-silicon detection rates are now very close to 100%.

Just enough to tread water ...

Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

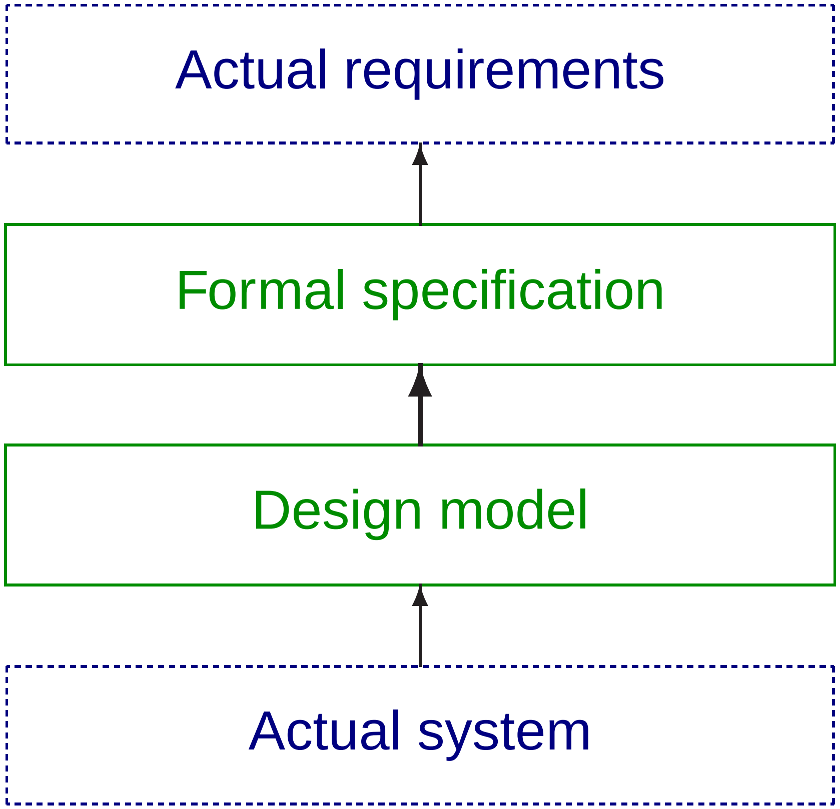
- Slow — especially pre-silicon
- Too many possibilities to test them all

For example:

- 2^{160} possible pairs of floating point numbers (possible inputs to an adder).
- Vastly higher number of possible states of a complex microarchitecture.

Formal verification

Formal verification: mathematically prove the correctness of a *design* with respect to a mathematical *formal specification*.



Proof vs. experiment

In principle, correctness of hardware or a program should be very much like a mathematical theorem. Yet:

- In mathematics it is normal to *prove* results rigorously, and experimental “inductive” testing is exotic and controversial.
- In computing, it is normal to establish results by empirical testing and proving them formally is exotic and controversial.

The value of proof

Sometimes an enormous weight of empirical evidence can be misleading.

- $\pi(n) - \int_0^n du/ln(u)$ does not change sign for $n \leq 10^{10}$
- The Pentium divider successfully ran billions of test cases and real applications

Yet in both cases the natural “inductive” conclusion was wrong.

Testing can miss things that would be revealed by a formal proof.

Formal verification in industry

Formal verification is increasingly becoming standard practice in the hardware industry. It is much less used in the software industry outside safety-critical niches.

Why the difference?

- Hardware is designed in a more modular way than most software.
- There is more scope for complete automation
- The potential consequences of a hardware error are greater

Formal verification methods

Many different methods are used in formal verification, mostly trading efficiency and automation against generality.

- Propositional tautology checking
- Symbolic simulation
- Symbolic trajectory evaluation
- Temporal logic model checking
- Decidable subsets of first order logic
- First order automated theorem proving
- Interactive theorem proving

Our work

Here we will focus on general interactive theorem proving.

We have formally verified correctness of various floating-point algorithms designed for the Intel® Itanium® architecture.

- Division
- Square root
- Transcendental functions (log, sin etc.)

Principia mathematical for the computer age?

Several pioneers showed how all proofs in ‘ordinary’ mathematics can be reduced to sequences of formulas in a precise formal system.

- Frege’s *Begriffsschrift*
- Peano’s *Rivista di Matematica*
- Russell and Whitehead’s *Principia Mathematica*

Unfortunately it’s extremely painful doing so by hand.

But with the aid of a computer, it’s much more palatable, and mistakes unlikely.

Theorem provers

There are several theorem provers that have been used for floating-point verification, some of it in industry:

- ACL2 (used at AMD)
- Coq
- HOL Light (used at Intel)
- PVS

All these are powerful systems with somewhat different strengths and weaknesses.

HOL Light overview

HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

HOL Light is designed to have a simple and clean logical foundation.

Versions written in CAML Light and Objective CAML.

Pushing the LCF approach to its limits

The main features of the LCF approach to theorem proving are:

- Reduce all proofs to a small number of relatively simple primitive rules
- Use the programmability of the implementation/interaction language to make this practical

Our work may represent the most “extreme” application of this philosophy.

- HOL Light’s primitive rules are very simple.
- Some of the proofs expand to about 100 million primitive inferences and can take many hours to check.

It is interesting to consider the scope of the LCF approach.

HOL Light primitive rules (1)

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x.\ t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \texttt{BETA}$$

HOL Light primitive rules (2)

$$\frac{}{\{p\} \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

HOL Light primitive rules (3)

$$\frac{\Gamma[x_1,\ldots,x_n] \vdash p[x_1,\ldots,x_n]}{\Gamma[t_1,\ldots,t_n] \vdash p[t_1,\ldots,t_n]}\ \texttt{INST}$$

$$\frac{\Gamma[\alpha_1,\ldots,\alpha_n] \vdash p[\alpha_1,\ldots,\alpha_n]}{\Gamma[\gamma_1,\ldots,\gamma_n] \vdash p[\gamma_1,\ldots,\gamma_n]}\ \texttt{INST\_TYPE}$$

Together with two definitional principles:

- for new constants equal to an existing term
- and new types in bijection with a nonempty set

Some of HOL Light’s derived rules

- Simplifier for (conditional, contextual) rewriting.
- Tactic mechanism for mixed forward and backward proofs.
- Tautology checker.
- Automated theorem provers for pure logic, based on tableaux and model elimination.
- Tools for definition of (infinitary, mutually) inductive relations.
- Tools for definition of (mutually) recursive datatypes
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.

Breakdown to primitive inferences

```
REAL_ARITH
  ‘a <= x /\ b <= y /\
   abs(x - y) < abs(x - a) /\
   abs(x - y) < abs(x - b) /\
   (b <= x ==> abs(x - a) <= abs(x - b)) /\
   (a <= y ==> abs(y - b) <= abs(y - a))
   ==> (a = b)‘;;
```

Takes 9.4 seconds (on my laptop) and generates 40040 primitive inferences.

Our HOL Light proofs

The mathematics we formalize is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

As part of the process, various special proof procedures for particular problems were programmed, e.g.

- Verifying solution set of some quadratic congruences
- Proving primality of particular numbers
- Proving bounds on rational approximations
- Verifying errors in polynomial approximations

LCF-style derived rules

How can we take a standard algorithm and produce a corresponding LCF-style derived rule? Usually some mixture of the following:

- Mimic each step of the algorithm, producing a theorem at each stage.

  Example: implement rewriting as a ‘conversion’ producing an equational theorem ($x = 2 \vdash x + 3 = 2 + 3$ etc.)

- Produce some ‘certificate’ and generate a formal proof in the checking process.

  Example: run some highly tuned first-order proof search and translate the proof eventually found.

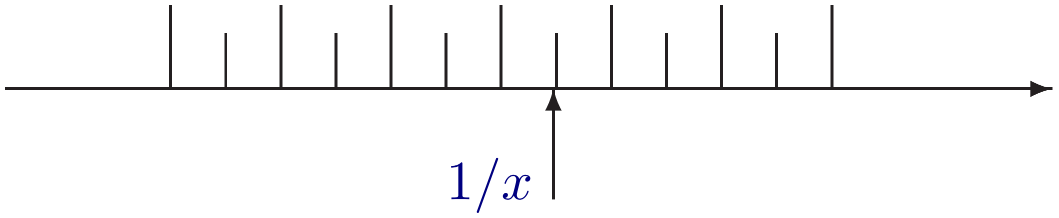
Second is also useful in connection with proof-carrying code.

Example: Difficult cases for reciprocals

Some algorithms for floating-point division, a/b, can be optimized for the special case of reciprocals ($a = 1$).

A direct analytic proof of the optimized algorithm is sometimes too hard because of the intricacies of rounding.

However, an analytic proof works for all but the ‘difficult cases’.



These are floating-point numbers whose reciprocal is very close to another one, or a midpoint, making them trickier to round correctly.

Mixed analytical-combinatorial proofs

By finding a suitable set of ‘diffifult cases’, one can produce a proof by a mixture of analytical reasoning and explicit checking.

- Find the set of difficult cases S
- Prove the algorithm analytically for all $x \notin S$
- Prove the algorithm by explicit case analysis for $x \in S$

Quite similar to some standard proofs in mathematics, e.g. Bertrand’s conjecture.

Finding difficult cases with factorization

After scaling to eliminate the exponents, finding difficult cases reduces to a straightforward number-theoretic problem.

A key component is producing the prime factorization of an integer *and* proving that the factors are indeed prime.

In typical applications, the numbers can be 49–227 bits long, so naive approaches based on testing all potential factors are infeasible.

The primality prover is embedded in a HOL derived rule `PRIME_CONV` that maps a numeral to a theorem asserting its primality or compositeness.

Certifying primality

We generate a ‘certificate of primality’ based on Pocklington’s theorem:

```
|- 2 ≤ n ∧
   (n - 1 = q * r) ∧
   n ≤ q EXP 2 ∧
   (a EXP (n - 1) == 1) (mod n) ∧
   (∀p. prime(p) ∧ p divides q
        ⇒ coprime(a EXP ((n - 1) DIV p) - 1,n))
   ⇒ prime(n)
```

The certificate is generated ‘extra-logically’, using the factorizations produced by PARI/GP.

The certificate is then checked by formal proof, using the above theorem.

The value of formal verification

The formal verifications we undertook had a number of beneficial consequences

- Uncovered several bugs
- Revealed ways that algorithms could be made more efficient
- Improved our confidence in the (original or final) algorithms
- Led to deeper theoretical understanding

This experience seems quite common.

Conclusions

- Formal verification is increasingly important in the hardware industry
- Some applications require general theorem proving
- LCF-style systems like HOL Light allow us to program various symbolic algorithms as reductions to primitive inferences
- These systems are difficult to use even for a specialist, and need further development to become more widely applicable.