Verifying floating-point algorithms using formalized mathematics

John Harrison
Intel Corporation

NYU Colloquium

February 6, 2004

The human cost of bugs

Computers are often used in safety-critical systems where a failure could cause loss of life.

- Heart pacemakers
- Aircraft
- Nuclear reactor controllers
- Car engine management systems
- Radiation therapy machines
- Telephone exchanges (!)
- ...

Financial cost of bugs

Even when not a matter of life and death, bugs can be financially serious if a faulty product has to be recalled or replaced.

- 1994 FDIV bug in the Intel®Pentium® processor: US $500 million.
- Today, new products are ramped much faster...

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.

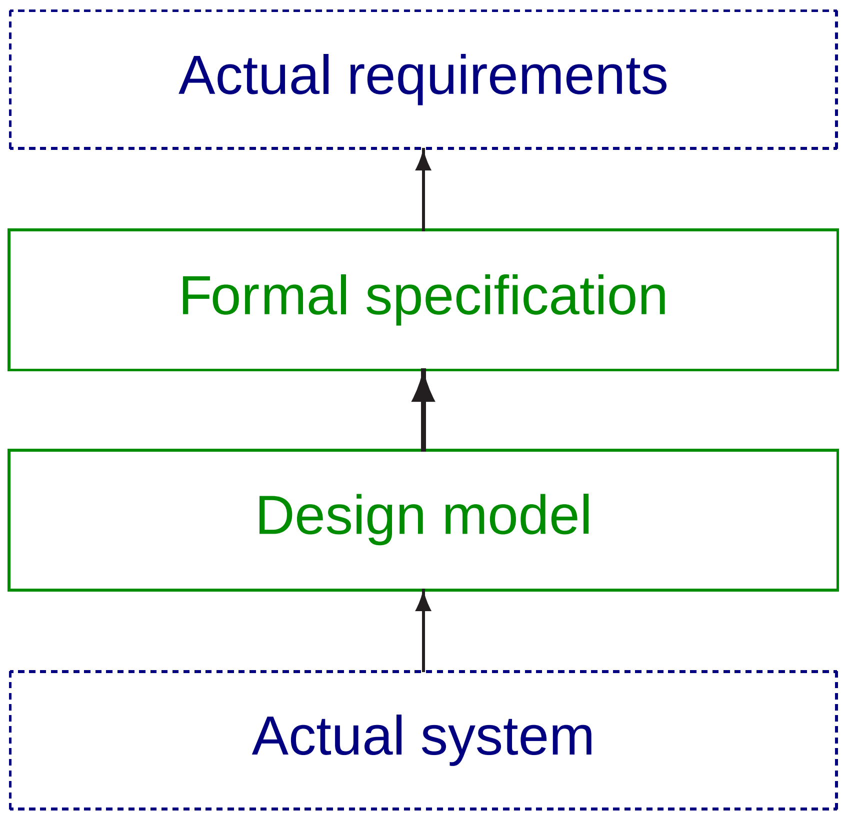
- Slow — especially pre-silicon
- Too many possibilities to test them all

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*.



Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

- Difficult and time-consuming.
- Only as reliable as the formal models used.
- How can we be sure the proof is right?

Analogy with mathematics

Sometimes even a huge weight of empirical evidence can be misleading.

- $\pi(n) =$ number of primes $\leq n$
- $li(n) = \int_0^n du/ln(u)$

Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often.

No change of sign at all had ever been found despite testing up to $n = 10^{10}$ (in the days before computers).

Similarly, extensive testing of hardware or software may still miss errors that would be revealed by a formal proof.

Formal verification is hard

Writing out a completely formal proof of correctness for real-world hardware and software is difficult.

- Must specify intended behaviour formally
- Need to make many hidden assumptions explicit
- Requires long detailed proofs, difficult to review

The state of the art is quite limited.

Software verification has been around since the 60s, but there have been few major successes.

Faulty hand proofs

“Synchronizing clocks in the presence of faults” (Lamport & Melliar-Smith, JACM 1985)

This introduced the Interactive Convergence Algorithm for clock synchronization, and presented a ‘proof’ of it.

- Presented five supporting lemmas and one main correctness theorem.
- Lemmas 1, 2, and 3 were all false.
- The proof of the main induction in the final theorem was wrong.
- The main result, however, was correct!

Machine-checked proof

A more promising approach is to have the proof checked (or even generated) by a computer program.

- It can reduce the risk of mistakes.
- The computer can automate some parts of the proofs.

There are limits on the power of automation, so detailed human guidance is often necessary.

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

From interactive proof checkers to fully automatic theorem provers.

AUTOMATH (de Bruijn)

Mizar (Trybulec)

...

PVS (Owre, Rushby, Shankar)

...

ACL2 (Boyer, Kaufmann, Moore)

Vampire (Voronkov)

Mathematical versus industrial

Some provers are intended to formalize pure mathematics, others to tackle industrial-scale verification

AUTOMATH (de Bruijn)

Mizar (Trybulec)

...

...

PVS (Owre, Rushby, Shankar)

ACL2 (Boyer, Kaufmann, Moore)

Our work

Here we will focus on general interactive theorem proving.

We have formally verified correctness of various floating-point algorithms for functions including:

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

The verifications are conducted using the HOL Light theorem prover.

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.

Floating point verification

We've used HOL Light to verify the accuracy of floating point algorithms (used in hardware and software) for:

- Division and square root
- Transcendental function such as sin, exp, $atan$.

This involves background work in formalizing:

- Real analysis
- Basic floating point arithmetic

Existing real analysis theory

- Definitional construction of real numbers
- Basic topology
- General limit operations
- Sequences and series
- Limits of real functions
- Differentiation
- Power series and Taylor expansions
- Transcendental functions
- Gauge integration

Examples of useful theorems

```
|- sin(x + y) = sin(x) * cos(y) + cos(x) * sin(y)

|- tan(&n * pi) = &0

|- &0 < x /\ &0 < y ==> (ln(x / y) = ln(x) - ln(y))

|- f contl x /\ g contl (f x) ==> (g o f) contl x

|- (!x. a <= x /\ x <= b ==> (f diffl (f' x)) x) /\
   f(a) <= K /\ f(b) <= K /\
   (!x. a <= x /\ x <= b /\ (f'(x) = &0) ==> f(x) <= K)
   ==> !x. a <= x /\ x <= b ==> f(x) <= K
```

HOL floating point theory (1)

We have formalized a floating point theory in HOL with the precision as a parameter.

A floating point format is identified by a triple of natural numbers `fmt`.

The corresponding set of real numbers is `format(fmt)`, or ignoring the upper limit on the exponent, `iformat(fmt)`.

Floating point rounding returns a floating point approximation to a real number, ignoring upper exponent limits. More precisely

```
round fmt rc x
```

returns the appropriate member of `iformat(fmt)` for an exact value `x`, depending on the rounding mode `rc`, which may be one of `Nearest`, `Down`, `Up` and `Zero`.

HOL floating point theory (2)

For example, the definition of rounding down is:

```
|- (round fmt Down x = closest
        {a | a IN iformat fmt ∧ a <= x} x)
```

We prove a large number of results about rounding, e.g.

```
|- ¬(precision fmt = 0) ∧ x IN iformat fmt
   ⇒ (round fmt rc x = x)
```

that rounding is monotonic:

```
|- ¬(precision fmt = 0) ∧ x <= y
   ⇒ round fmt rc x <= round fmt rc y
```

and that subtraction of nearby floating point numbers is exact:

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   a / &2 <= b ∧ b <= &2 * a ⇒ (b - a) IN iformat fmt
```

The $(1 + \epsilon)$ property

Designers often rely on clever "cancellation" tricks to avoid or compensate for rounding errors.

But many routine parts of the proof can be dealt with by a simple conservative bound on rounding error:

```
|- normalizes fmt x ∧
   ¬(precision fmt = 0)
   ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
           (round fmt rc x = x * (&1 + e))
```

Derived rules apply this result to computations in a floating point algorithm automatically, discharging the conditions as they go.

Example: tangent algorithm

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.
- If the reduced argument r is still not small enough, it is separated into its leading few bits B and the trailing part $x = r - B$, and the overall result computed from $tan(x)$ and pre-stored functions of B, e.g.

$$tan(B + x) = tan(B) + \frac{\frac{1}{sin(B)cos(B)} tan(x)}{cot(B) - tan(x)}$$

- Now a power series approximation is used for $tan(r)$, $cot(r)$ or $tan(x)$ as appropriate.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.
- The rounding errors involved in computing with floating point arithmetic are within bounds.

Most of these parts are non-trivial. Moreover, some of them require more pure mathematics than might be expected.

Why mathematics?

Controlling the error in range reduction becomes difficult when the reduced argument $X - N\pi/2$ is small.

To check that the computation is accurate enough, we need to know:

> How close can a floating point number be to an integer multiple of $\pi/2$?

Even deriving the power series (for $0 < |x| < \pi$):

$$cot(x) = 1/x - \frac{1}{3}x - \frac{1}{45}x^3 - \frac{2}{945}x^5 - \ldots$$

is much harder than you might expect.

Polynomial approximation errors

Many transcendental functions are ultimately approximated by polynomials in this way.

This usually follows some initial reduction step to ensure that the argument is in a small range, say $x \in [a, b]$.

The *minimax* polynomials used have coefficients found numerically to minimize the maximum error over the interval.

In the formal proof, we need to prove that this is indeed the maximum error, say $\forall x \in [a, b]. |sin(x) - p(x)| \leq 10^{-62}|x|$.

By using a Taylor series with much higher degree, we can reduce the problem to bounding a pure polynomial with rational coefficients over an interval.

Bounding functions

If a function f differentiable for $a \le x \le b$ has the property that $f(x) \le K$ at all points of zero derivative, as well as at $x = a$ and $x = b$, then $f(x) \le K$ everywhere.

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl (f' x)) x) ∧
   f(a) <= K ∧ f(b) <= K ∧
   (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0)
        ⇒ f(x) <= K)
   ⇒ (∀x. a <= x ∧ x <= b ⇒ f(x) <= K)
```

Hence we want to be able to isolate zeros of the derivative (which is just another polynomial).

Isolating derivatives

For any differentiable function f, $f(x)$ can be zero only at one point between zeros of the derivative $f'(x)$.

More precisely, if $f'(x) \neq 0$ for $a < x < b$ then if $f(a)f(b) \geq 0$ there are no points of $a < x < b$ with $f(x) = 0$:

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl f'(x))(x)) ∧
   (∀x. a < x ∧ x < b ⇒ ¬(f'(x) = &0)) ∧
   f(a) * f(b) >= &0
   ⇒ ∀x. a < x ∧ x < b ⇒ ¬(f(x) = &0)
```

Bounding and root isolation

This gives rise to a recursive procedure for bounding a polynomial and isolating its zeros, by successive differentiation.

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl (f' x)) x) ∧
   (∀x. a <= x ∧ x <= b ⇒ (f' diffl (f'' x)) x) ∧
   (∀x. a <= x ∧ x <= b ⇒ abs(f''(x)) <= K) ∧
   a <= c ∧ c <= x ∧ x <= d ∧ d <= b ∧ (f'(x) = &0)
   ⇒ abs(f(x)) <= abs(f(d)) + (K / &2) * (d - c) pow 2
```

At each stage we actually produce HOL theorems asserting bounds and the enclosure properties of the isolating intervals.

Conclusions

- Formal verification is industrially important, and can be attacked with current theorem proving technology.
- A large part of our work involves building up general theories about both pure mathematics and special properties of floating point numbers.
- It is easy to underestimate the amount of pure mathematics needed for obtaining very practical results.
- The mathematics required is often the sort that is not found in current textbooks: very concrete results but with a proof!
- Using HOL Light, we can confidently integrate all the different aspects of the proof, using programmability to automate tedious parts.