

# Mathematical Modeling to Formally Prove Correctness

---

John Harrison  
Intel Corporation

Gelato Meeting

April 24, 2006

## 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, the consequences of bugs can be quite dramatic.

- In 1996, the Ariane 5 rocket made its first flight
- It was automatically destroyed shortly after takeoff
- The cause was an uncaught exception on floating-point to integer conversion.

## Another floating-point bug

---

Intel has also had at least one major floating-point issue:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US \$475 million to cover the costs.

## Things are not getting easier

---

The environment is becoming even less benign:

- The overall market is much larger, so the potential cost of recall/replacement is far higher.
- New products are ramped faster and reach high unit sales very quickly.
- Competitive pressures are leading to more design complexity.

## Some complexity metrics

---

Recent Intel processor generations (Pentium, P6 and Pentium 4) indicate:

- A 4-fold increase in overall complexity (lines of RTL ...) per generation
- A 4-fold increase in design bugs 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 keep our heads above 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

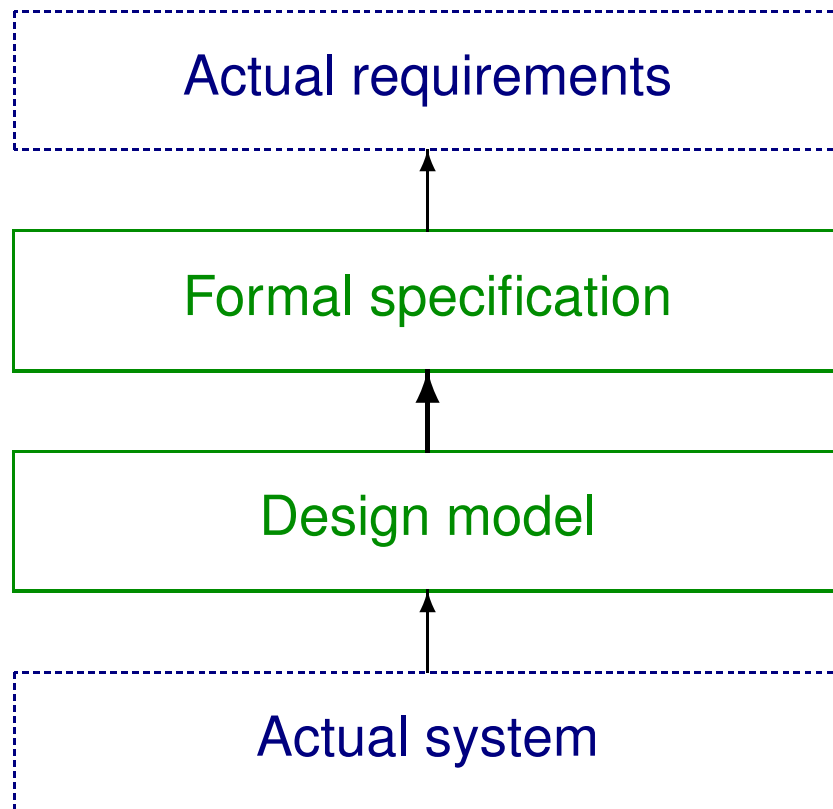
For example:

- $2^{160}$  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.

## 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.

## A spectrum of formal techniques

---

There are various possible levels of rigor in correctness proofs:

- Programming language typechecking
- Lint-like static checks (uninitialized variables ...)
- Checking of loop invariants and other annotations
- Complete functional verification

## FV in the software industry

---

Some recent success with partial verification in the software world:

- Analysis of Microsoft Windows device drivers using SLAM
- Non-overflow proof for Airbus A380 flight control software

Much less use of full functional verification. Very rare except in highly safety-critical or security-critical niches.

## FV in the hardware industry

---

In the hardware industry, full functional correctness proofs are increasingly becoming common practice.

- 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

---

We will focus on our own formal verification activities:

- Formal verification of floating-point operations
- Targeted at the Intel® Itanium® processor family.
- Conducted using the interactive theorem prover HOL Light.

## Why floating-point?

---

There are obvious reasons for focusing on floating-point:

- Known to be difficult to get right, with several issues in the past.  
**We don't want another FDIV!**
- Quite clear specification of how most operations *should* behave.  
**We have the IEEE Standard 754.**

However, Intel is also applying FV in many other areas, e.g. control logic, cache coherence, bus protocols . . .

## Why interactive theorem proving?

---

Limited scope for highly automated finite-state techniques like model checking.

It's difficult even to specify the intended behaviour of complex mathematical functions in bit-level terms.

We need a general framework to reason about mathematics in general while checking against errors.

## 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  $\lambda$ -calculus.

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

Current version written in Objective CAML ("OCaml").

## What does LCF mean?

---

The name is a historical accident:

The original Stanford and Edinburgh LCF systems were for Scott's Logic of Computable Functions.

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

## No free lunch

---

There is no practical way of automatically proving highly sophisticated mathematics.

Some isolated successes such as the solution of the Robbins conjecture . . .

Mostly, we content ourselves with automating “routine” parts of the proof.

## Automating the routine

---

We can automate linear inequality reasoning:

$$\begin{aligned} & a \leq x \wedge b \leq y \wedge |x - y| < |x - a| \wedge |x - y| < |x - b| \wedge \\ & (b \leq x \Rightarrow |x - a| < |x - b|) \wedge (a \leq y \Rightarrow |y - b| < |x - a|) \\ & \Rightarrow a = b \end{aligned}$$

and basic algebraic rearrangement:

$$\begin{aligned} & (w_1^2 + x_1^2 + y_1^2 + z_1^2) \cdot (w_2^2 + x_2^2 + y_2^2 + z_2^2) = \\ & (w_1 \cdot w_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2)^2 + \\ & (w_1 \cdot x_2 + x_1 \cdot w_2 + y_1 \cdot z_2 - z_1 \cdot y_2)^2 + \\ & (w_1 \cdot y_2 - x_1 \cdot z_2 + y_1 \cdot w_2 + z_1 \cdot x_2)^2 + \\ & (w_1 \cdot z_2 + x_1 \cdot y_2 - y_1 \cdot x_2 + z_1 \cdot w_2)^2 \end{aligned}$$

## The obviousness mismatch

---

Can also automate some purely logical reasoning such as this:

$$\begin{aligned} & (\forall x y z. P(x, y) \wedge P(y, z) \Rightarrow P(x, z)) \wedge \\ & (\forall x y z. Q(x, y) \wedge Q(y, z) \Rightarrow Q(x, z)) \wedge \\ & (\forall x y. Q(x, y) \Rightarrow Q(y, x)) \wedge \\ & (\forall x y. P(x, y) \vee Q(x, y)) \\ & \Rightarrow (\forall x y. P(x, y)) \vee (\forall x y. Q(x, y)) \end{aligned}$$

As Łoś points out, this is not obvious for most people.

## 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

---

$$\vdash \sin(x + y) = \sin(x) * \cos(y) + \cos(x) * \sin(y)$$

$$\vdash \tan(n * \pi) = 0$$

$$\vdash 0 < x \wedge 0 < y \Rightarrow (\ln(x / y) = \ln(x) - \ln(y))$$

$$\vdash f \text{ contl } x \wedge g \text{ contl } (f \ x) \Rightarrow (g \circ f) \text{ contl } x$$

$$\vdash (\forall x. a \leq x \wedge x \leq b \Rightarrow (f \text{ diff1 } (f' \ x)) \ x) \wedge f(a) \leq K \wedge f(b) \leq K \wedge$$

$$(\forall x. a \leq x \wedge x \leq b \wedge (f'(x) = 0) \Rightarrow f(x) \leq K)$$

$$\Rightarrow \forall x. a \leq x \wedge x \leq b \Rightarrow f(x) \leq K$$

## HOL floating point theory (1)

---

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:

$$\begin{aligned} &|- \text{ (round fmt Down } x = \text{closest} \\ &\quad \{a \mid a \text{ IN iformat fmt } \wedge a \leq x\} x) \end{aligned}$$

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

$$\begin{aligned} &|- \neg(\text{precision fmt} = 0) \wedge x \text{ IN iformat fmt} \\ &\quad \Rightarrow \text{ (round fmt rc } x = x) \end{aligned}$$

that rounding is monotonic:

$$\begin{aligned} &|- \neg(\text{precision fmt} = 0) \wedge x \leq y \\ &\quad \Rightarrow \text{ round fmt rc } x \leq \text{ round fmt rc } y \end{aligned}$$

and that subtraction of nearby floating point numbers is exact:

$$\begin{aligned} &|- a \text{ IN iformat fmt } \wedge b \text{ IN iformat fmt } \wedge \\ &\quad a / \&2 \leq b \wedge b \leq \&2 * a \Rightarrow (b - a) \text{ IN iformat fmt} \end{aligned}$$

## 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  $0 < |x| < \pi$ ):

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

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

$$\begin{aligned} &|- (\forall x. a \leq x \wedge x \leq b \Rightarrow (f \text{ diff1 } (f' \ x)) \ x) \wedge \\ & f(a) \leq K \wedge f(b) \leq K \wedge \\ & (\forall x. a \leq x \wedge x \leq b \wedge (f'(x) = 0) \\ & \quad \Rightarrow f(x) \leq K) \\ & \Rightarrow (\forall x. a \leq x \wedge x \leq b \Rightarrow f(x) \leq K) \end{aligned}$$

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  $f(x) = 0$ :

$$\begin{aligned} &|- (\forall x. a \leq x \wedge x \leq b \Rightarrow (f \text{ diff1 } f'(x))(x)) \wedge \\ & (\forall x. a < x \wedge x < b \Rightarrow \neg(f'(x) = 0)) \wedge \\ & f(a) * f(b) \geq 0 \\ & \Rightarrow \forall x. a < x \wedge x < b \Rightarrow \neg(f(x) = 0) \end{aligned}$$

## Bounding and root isolation

---

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

$$\begin{aligned} &|- (\forall x. a \leq x \wedge x \leq b \Rightarrow (f \text{ diff1 } (f' \ x)) \ x) \wedge \\ &(\forall x. a \leq x \wedge x \leq b \Rightarrow (f' \text{ diff1 } (f'' \ x)) \ x) \wedge \\ &(\forall x. a \leq x \wedge x \leq b \Rightarrow \text{abs}(f'' \ (x)) \leq K) \wedge \\ &a \leq c \wedge c \leq x \wedge x \leq d \wedge d \leq b \wedge (f' \ (x) = 0) \\ &\Rightarrow \text{abs}(f(x)) \leq \text{abs}(f(d)) + (K / 2) * (d - c) \text{ pow } 2 \end{aligned}$$

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.