High-Level Verification using Theorem Proving and Formalized Mathematics

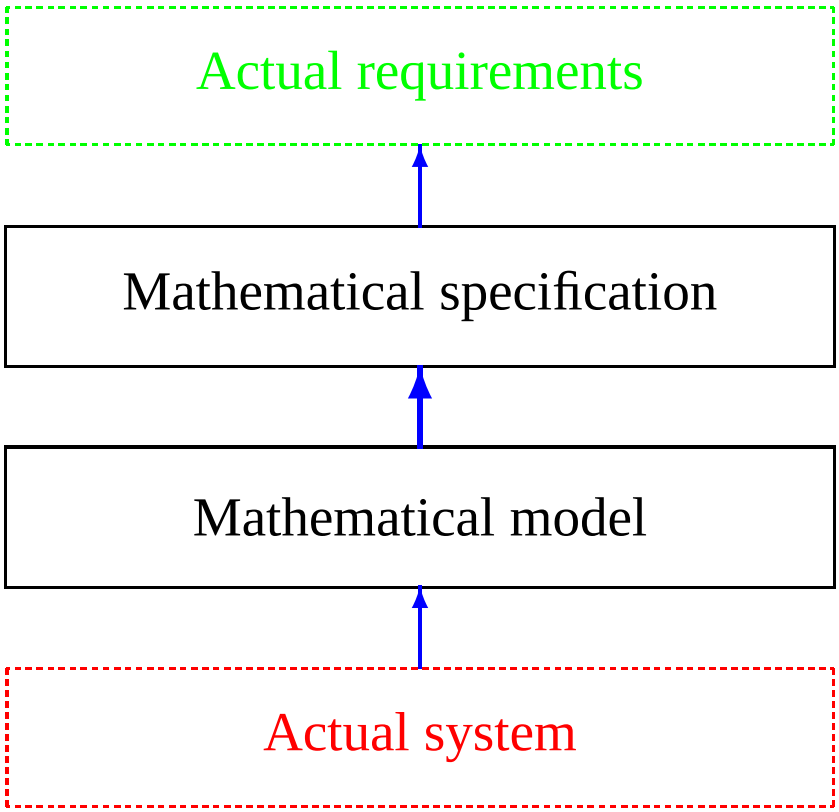
John Harrison

Intel Corporation

- A hierarchy of verifications
- What do we need?
- HOL Light
- Theories of reals and floating point numbers
- Tangent algorithm
- Mathematics of range reduction and power series
- Conclusions

Formal verification and formal models

Formal verification aims to make a link (by rigorous mathematical proof) between a mathematical model of an actual system and a model of its intended behavior.



A hierarchy of verifications

Experience shows that computer systems can better be kept intellectually manageable if designed hierarchically, with clear interfaces between layers.

Formal verifications should be structured in the same way, i.e. requirements at one level should be the system model at the next level up. Consider the precise mathematical formalization of:

> The execution of the `FADD` instruction results in writing to the appropriate register the IEEE-specified floating point sum of the contents of the input registers . . .

For someone (Carl) verifying the correctness of the hardware implementation, this is part of the specification to be proved.

For someone (John) verifying correctness of mathematical software that uses operations like `FADD`, this is one of the assumptions in the implementation model.

Verification of mathematical software

We are mostly concerned with software for calculating the common mathematical functions, e.g. $tan(x)$ for an input number x, and formally verifying its numerical accuracy.

We will assume that all the operations used obey the underlying specifications as given in the Architecture Manual and the IEEE Standard for Binary Floating-Point Arithmetic.

What do we need to formally verify such mathematical software?

- Theorems about basic real analysis and properties of the transcendental functions.
- Theorems about special properties of floating point numbers, floating point rounding etc.
- Automation of as much tedious reasoning as possible
- A flexible framework in which these components can be developed and applied in a reliable way.

The spectrum of theorem provers

From interactive proof checkers to fully automatic theorem provers.

AUTOMATH (de Bruijn)

Stanford LCF (Milner)

Mizar (Trybulec)

. . .

. . .

PVS (Owre, Rushby, Shankar)

. . .

. . .

ACL2 (Boyer, Kaufmann, Moore)

Otter (McCune)

Quick introduction to HOL Light

HOL Light is a member of the family of HOL theorem provers.

- An LCF-style programmable proof checker written in CAML Light, which also serves as the interaction language.
- Supports classical higher order logic based on polymorphic simply typed lambda-calculus.
- Extremely simple logical core: 10 basic logical inference rules plus 2 definition mechanisms and 3 axioms.
- More powerful proof procedures programmed on top, inheriting their reliability from the logical core. Fully programmable by the user.
- Well-developed mathematical theories including basic real analysis.

HOL Light is available for download from:

```
http://www.cl.cam.ac.uk/users/jrh/hol-light
```

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)

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

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

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

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

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

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 10.6 seconds (on my laptop) and generates 40040 primitive inferences:

REFL	11466
TRANS	4429
MK_COMB	6057
ABS	0
BETA	1989
ASSUME	288
EQ_MP	7536
DEDUCT_ANTISYM_RULE	1882
INST_TYPE	479
INST	5914
TOTAL	**40040**

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)
   ==> (\x. g(f x)) 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

We have formalized a generic floating point theory in HOL, which can be applied to all the required formats, and others supported in software such as quad precision.

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`.

The $(1+\varepsilon)$ property

Most of the routine parts of floating point proofs rely on either an absolute or relative bound on the effect of floating point rounding. The key theorem underlying relative error analysis is the following:

```
|- normalizes fmt x /\
   ~(precision fmt = 0)
   ==> ?e. abs(e) <= mu rc /
                  &2 pow (precision fmt - 1) /\
              (round fmt rc x = x * (&1 + e))
```

This says that given that the value being rounded is in the range of normalized floating point numbers, then rounding perturbs the exact result by at most a relative error bound depending only on the floating point precision and rounding control.

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

Cancellation theorems

Low-level mathematical algorithms often rely on special tricks to avoid rounding error, or compensate for it. Rounding is trivial when the value being rounded is already representable exactly:

```
|- a IN iformat fmt ==> (round fmt rc a = a)
```

Some special situations where this happens are as follows:

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

|- x IN iformat fmt /\
   y IN iformat fmt /\
   abs(x) <= abs(y)
   ==> (round fmt Nearest (x + y) - y)
                  IN iformat fmt /\
       (round fmt Nearest (x + y) - (x + y))
                  IN iformat fmt
```

A tangent algorithm

An algorithm to calculate tangents works essentially as follows.

- 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

In order 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.
- The pre-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. We will look at the mathematics required to analyze range reduction and the power series approximation.

Range reduction (1)

Range reduction involves a fairly complicated computation, using various tricks to avoid rounding error. This can mostly be dealt with using the general lemmas given above. However, controlling the errors is harder the smaller the reduced argument is, so we need to answer the key mathematical question:

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

To answer this question, we need to formalize in HOL some theorems about rational approximations. First of all, we have formalized some results allowing us to (provably) find arbitrarily good rational approximations to π, e.g. the series:

$$\pi = \Sigma_{m=0}^{\infty} \frac{1}{16^m}\left(\frac{4}{8m+1} - \frac{2}{8m+4} - \frac{1}{8m+5} - \frac{1}{8m+6}\right)$$

Range reduction (2)

We then formalize the proof that *convergents* to a real number x, i.e. rationals $p_1/q_1 < x < p_2/q_2$ with $p_2 q_1 = p_1 q_2 + 1$, are the best possible approximation without having a larger denominator.

```
|- (p2 * q1 = p1 * q2 + 1) /\
   (&p1 / &q1 < x /\ x < &p2 / &q2)
   ==> !b. ~(b = 0) /\ b < q1 /\ b < q2
           ==> abs(&a / &b - x)
                 > &1 / &(q1 * q2)
```

We find such convergents (outside the logic) using the Stern-Brocot tree, and by inserting the values into the approximation theorems, and can answer the above question for input numbers in the specified range:

```
|- integer(N) /\ ~(N = &0) /\
   a IN iformat (rformat Register) /\
   abs(a) < &2 pow 64
   ==> abs (a - N * pi / &2)
           >= &113 / &2 pow 76
```

Power series approximation (1)

The power series for tangent and cotangent are found in many mathematical handbooks. For example (for $x \neq 0$):

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

However, such handbooks typically don't give any proof, while more rigorous works don't usually discuss such concrete results at all. It's no accident that the proof we eventually found and formalized is in an older book: Knopp's *"Infinite Series"*. By a rather complicated limit argument we can prove:

$$\pi x\, cot(\pi x) = 1 + 2x^2 \Sigma_{k=1}^{\infty} \frac{1}{x^2 - k^2}$$

Power series approximation (2)

We can then expand the individual terms of the power series:

$$\frac{-x^2}{x^2-k^2} = \Sigma_{n=1}^{\infty}(x^2/k^2)^n$$

Since all terms have the same sign, it's fairly easy to show that we can reverse the order of the summations. This gives us a power series with coefficients expressed in terms of the harmonic sums like $1+1/2^4+1/3^4+1/4^4+\cdots$. By using the fact that $cot(x)-2cot(2x)=tan(x)$ (for $0<|x|<\pi/2$), we can compare the coefficients against the derivatives of tan and hence get them as rational numbers. As a byproduct, we derive various well-known theorems like:

$$\begin{aligned} 1+1/2^2+1/3^2+1/4^2+\cdots &= \pi^2/6 \\ 1+1/2^4+1/3^4+1/4^4+\cdots &= \pi^4/90 \end{aligned}$$

Conclusions

- Formal verification of mathematical software is industrially important, and can be attacked with current theorem proving technology.
- A large part of the 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.
- These proofs are probably the largest ever formally generated in such a simple primitive inference system.