Floating-Point Verification by Theorem Proving

John Harrison

Intel Corporation

Capital Normal University

13th October 2012

Overview

- Famous computer arithmetic failures
- Formal verification and theorem proving
- Floating-point arithmetic
- Division and square root
- Transcendental functions

Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot
- Internal clock was multiplied by $\frac{1}{10}$ to produce time in seconds
- Actually performed by multiplying 24-bit approximation of $\frac{1}{10}$
- Net error after 100 hours about 0.34 seconds.
- A Scud missile travels $500m$ in that time

Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth $\$500M$.

- Cause was an uncaught floating-point exception
- A 64-bit floating-point number repreenting horizontal velocity was converted to a 16-bit integer
- The number was larger than 2^{15}.
- As a result, the conversion failed.
- The rocket veered off its flight path and exploded, just 40 seconds into the flight sequence.

Vancouver stock exchange

In 1982 the Vancouver stock exchange index was established at a level of 1000.

A couple of years later the index was hitting lows of around 520.

The cause was repeated truncation of the index to 3 decimal digits on each recalculation, several thousand times a day.

On correction, the stock index leapt immediately from 574.081 to 1098.882.

A floating-point bug closer to home

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.

Remember the HP-35?

The Hewlett-Packard HP-35 calculator (1972) also had floating-point bugs:

- Exponential function, e.g. $e^{ln(2.02)} = 2.00$
- sin of some small angles completely wrong

At this time HP had already sold 25,000 units, but they advised users of the problem and offered a replacement:

> “We’re going to tell everyone and offer them, a replacement.
> It would be better to never make a dime of profit than to have
> a product out there with a problem.” (Dave Packard.)

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

Study of evolution of complexity from Pentium through 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.

So Intel is very active in formal verification.

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

Intel’s formal verification work

Intel uses formal verification quite extensively, e.g.

- Verification of Intel® Pentium® 4 floating-point unit with a mixture of STE and theorem proving
- Verification of bus protocols using pure temporal logic model checking
- Verification of microcode and software for many Intel® Itanium® floating-point operations, using pure theorem proving

FV found many high-quality bugs in P4 and verified “20%” of design

FV is now standard practice in the floating-point domain

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?

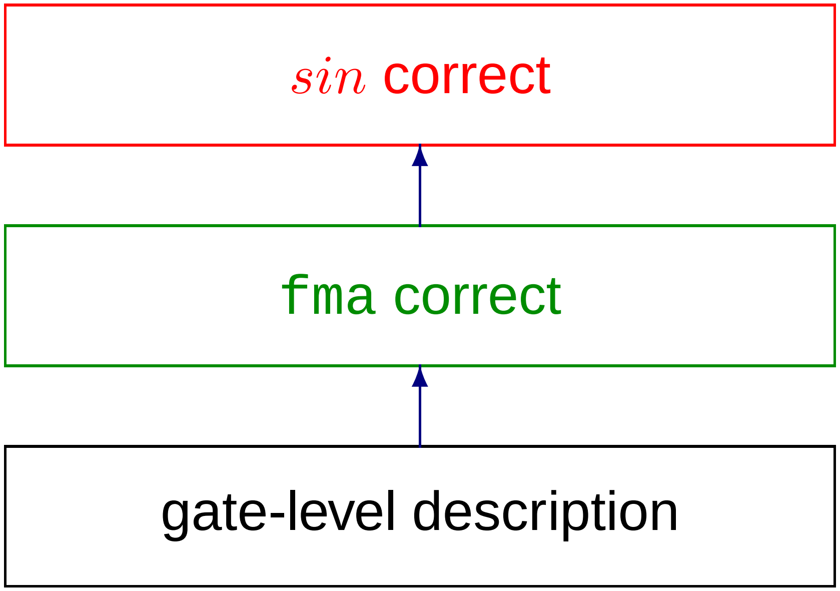
Limited scope for highly automated finite-state techniques like model checking.

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.

Levels of verification

High-level algorithms assume correct behavior of some hardware primitives.



sin correct

`fma` correct

gate-level description

Proving my assumptions is someone else's job . . .

Characteristics of this work

The verification we're concerned with is somewhat atypical:

- Rather simple according to typical programming metrics, e.g. 5-150 lines of code, often no loops.
- Relies on non-trivial mathematics including number theory, analysis and special properties of floating-point rounding.

Tools that are often effective in other verification tasks, e.g. temporal logic model checkers, are of almost no use.

What do we need?

We need a general theorem proving system with:

- Ability to mix interactive and automated proof
- Programmability for domain-specific proof tasks
- A substantial library of pre-proved mathematics

HOL Light satisfies these requirements nicely.

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.
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.
- Generic normalizers for rings and fields
- General quantifier elimination over $\mathbb{C}$
- Gröbner basis algorithm over fields

Automating the routine

We can automate linear inequality reasoning:

$$a \leq x \land b \leq y \land |x - y| < |x - a| \land |x - y| < |x - b| \land$$
$$(b \leq x \Rightarrow |x - a| < |x - b|) \land (a \leq y \Rightarrow |y - b| < |x - a|)$$
$$\Rightarrow a = b$$

and basic algebraic rearrangement:

$$(w_1^2 + x_1^2 + y_1^2 + z_1^2) \cdot (w_2^2 + x_2^2 + y_2^2 + z_2^2) =$$
$$(w_1 \cdot w_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2)^2 +$$
$$(w_1 \cdot x_2 + x_1 \cdot w_2 + y_1 \cdot z_2 - z_1 \cdot y_2)^2 +$$
$$(w_1 \cdot y_2 - x_1 \cdot z_2 + y_1 \cdot w_2 + z_1 \cdot x_2)^2 +$$
$$(w_1 \cdot z_2 + x_1 \cdot y_2 - y_1 \cdot x_2 + z_1 \cdot w_2)^2$$

The obviousness mismatch

Can also automate some purely logical reasoning such as this:

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

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

Real analysis details

Real analysis is especially important in our applications

- 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

Floating point numbers

Usually, the floating point numbers are those representable in some number n of significant binary digits, within a certain exponent range:

$$(-1)^s \times d_0.d_1d_2\cdots d_n \times 2^e$$

where

- $s \in \{0,1\}$ is the *sign*
- $d_0.d_1d_2\cdots d_n$ is the *significand* and $d_1d_2\cdots d_n$ is the *fraction* (aka mantissa).
- e is the exponent.

We often refer to $p = n + 1$ as the *precision*.

HOL floating point formats

We have formalized a generic floating point theory in HOL, which can be applied to all hardware formats, and others supported in software.

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).

```
 |- iformat (E,p,N) =
      { x | ∃s e k. s < 2 ∧ k < 2 EXP p ∧
                   x = --(&1) pow s * &2 pow e *
                       &k / &2 pow N}
```

We distinguish carefully between actual floating point numbers (as bitstrings) and the corresponding real numbers.

Supported formats

The Intel® Itanium® architecture supports many floating-point formats including:

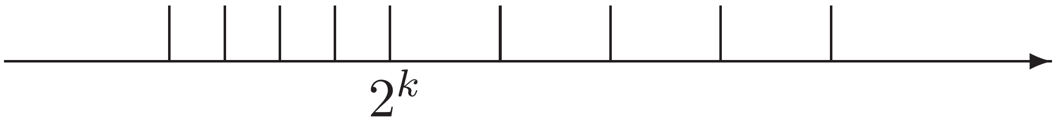
Format name	p	E_{min}	E_{max}
Single	24	-126	127
Double	53	-1022	1023
Double-extended	64	-16382	16383
Register	64	-65534	65535

The double-extended format, as well as some additional “mixed” formats, support compatibility with IA-32.

Units in the last place

It’s customary to give a bound on the error in transcendental functions in terms of ‘units in the last place’ (ulps).

While ulps are a standard way of measuring error, there’s a remarkable lack of unanimity in published definitions of the term. One of the merits of a formal treatment is to clear up such ambiguities.



Roughly, a unit in the last place is the gap between adjacent floating point numbers. But at the boundary 2^k between ‘binades’, this distance changes.

Two definitions

Goldberg considers the binade containing the computed result:

> In general, if the floating-point number $d.d \cdots d \times \beta^e$ is used to represent z, it is in error by $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}e$ units in the last place.

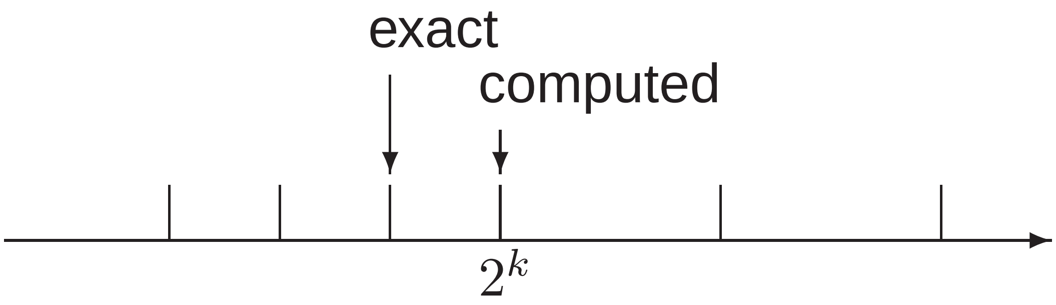
Muller considers the binade containing the exact result:

> The term *ulp(x) (*for *unit in the last place*) denotes the distance between the two floating point numbers that are closest to x.

However these both have counterintuitive properties.

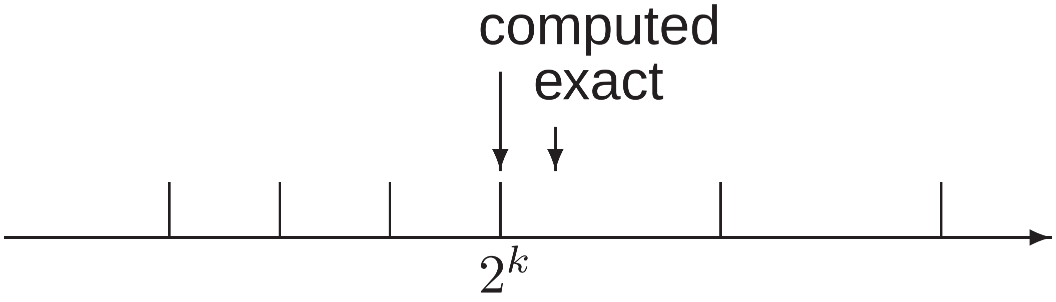
Problems with these definitions (1)

An error of $0.5ulp$ according to Goldberg, but intuitively $1ulp$.



Problems with these definitions (2)

An error of $0.4ulp$ according to Muller, but intuitively 0.2 ulp. Rounding up is worse...



Our definition: $ulp(x)$ is the distance between the closest pair of floating point numbers a and b with $a \leq x \leq b$. Note that we are counting the exact result 2^k as belonging to the binade *below*.

IEEE-correct operations

The IEEE Standard 754 specifies for all the usual algebraic operations, including square root but *not* transcendentals:

> . . . each of these operations shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range and then coerced this intermediate result to fit in the destination's format [using the specified rounding operation]

The Intel® Itanium® architecture extends the set of IEEE-correct operations with a ternary multiply-add $x \cdot y + z$ (and $x \cdot y - z$ and $z - x \cdot y$.

Rounding (1)

Rounding is controlled by a rounding mode, which is defined in HOL as an enumerated type:

```
 roundmode = Nearest | Down | Up | Zero
```

We define notions of ‘closest approximation’ as follows:

```
|- is_closest s x a =
       a IN s ∧ ∀b. b IN s ⇒ abs(b - x) >= abs(a - x)

|- closest s x = εa. is_closest s x a

|- closest_such s p x =
      εa. is_closest s x a ∧
             (∀b. is_closest s x b ∧ p b ⇒ p a)
```

Rounding (2)

Hence the actual definition of rounding:

```
 |- (round fmt Nearest x =
        closest_such (iformat fmt)
                     (EVEN o decode_fraction fmt) x) ∧
    (round fmt Down x =
        closest {a | a IN iformat fmt ∧ a <= x} x) ∧
    (round fmt Up x =
        closest {a | a IN iformat fmt ∧ a >= x} x) ∧
    (round fmt Zero x =
        closest {a | a IN iformat fmt ∧ abs a <= abs x} x)
```

Note that this is almost a direct transcription of the standard; no need to talk about ulps etc.

But it is also completely non-constructive!

Theorems about rounding

We prove some basic properties of rounding, e.g. that an already-representable number rounds to itself and conversely:

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

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

and that rounding is monotonic in all rounding modes:

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

There are various other simple properties, e.g. symmetries and skew-symmetries like:

```
|- ¬(precision fmt = 0)
   ⇒ (round fmt Down (--x) = --(round fmt Up x))
```

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.

Exact calculation

A famous theorem about exact calculation:

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

The following shows how we can retrieve the rounding error in multiplication using a fused multiply-add.

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a * b)
   ⇒ (a * b - round fmt Nearest (a * b)) IN iformat fmt
```

Here’s a similar one for addition and subtraction:

```
|- 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
```

Proof tools and execution

Several definitions are highly non-constructive, notably rounding.

However, we can prove equivalence to a constructive definition and hence prove particular numerical results:

```
#ROUND_CONV ‘round (10,11,12) Nearest (&22 / &7)‘;;
|- round (10,11,12) Nearest (&22 / &7) = &1609 / &512
```

Internally, HOL derives this using theorems about sufficient conditions for correct rounding.

In ACL2, we would be forced to adopt a non-standard constructive definition, but would then have such proving procedures without further work and highly efficient.

Division and square root

There are several different algorithms for division and square root, and which one is better is a fine choice.

- Digit-by-digit: analogous to pencil-and-paper algorithms but usually with quotient estimation and redundant digits (SRT, Ercegovac-Lang etc.)
- Multiplicative: get faster (e.g. quadratic) convergence by using multiplication, e.g. Newton-Raphson, Goldschmidt, power series.

The Intel® Itanium® architecture uses some interesting multiplicative algorithms relying *purely* on conventional floating-point operations.

Basic ideas due to Peter Markstein, first used in IBM Power series.

Correctness issues

Easy to get within a bit or so of the right answer, but meeting the IEEE spec is significantly more challenging.

In addition, all the flags need to be set correctly, e.g. inexact, underflow, ...

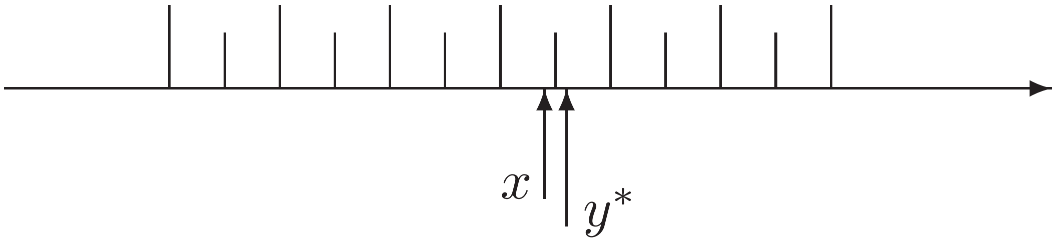
Whatever the overall structure of the algorithm, we can consider its last operation as yielding a result y by rounding an exact value y^*. What is the required property for perfect rounding?

We will concentrate on round-to-nearest mode.

Condition for perfect rounding

Sufficient condition for perfect rounding: the closest floating point number to the exact answer x is also the closest to y^*, the approximate result before the last rounding. That is, the two real numbers x and y^* never fall on opposite sides of a midpoint between two floating point numbers.

In the following diagram this is not true; x would round to the number below it, but y^* to the number above it.



How can we prove this?

Proving perfect rounding

There are two distinct approaches to justifying perfect rounding:

- Specialized theorems that analyze the precise way in which the approximation y^* rounds and how this relates to the mathematical function required.
- More direct theorems that are based on general properties of the function being approximated.

We will demonstrate how both approaches have been formalized in HOL.

- Verification of division algorithms based on a special technique due to Peter Markstein.
- Verification of square root algorithms based on an ‘exclusion zone’ method due to Marius Cornea

Markstein's main theorem

Markstein proves that given a quotient approximation $q_0 \approx \frac{a}{b}$ and a reciprocal approximation $y_0 \approx \frac{1}{b}$ with:

- The approximation q_0 is within $1\ ulp$ of $\frac{a}{b}$.
- The reciprocal approximation y_0 is $\frac{1}{b}$ rounded to the nearest floating point number

then if we execute the following two `fma` (fused multiply add) operations:

$$\begin{aligned} r &= a - bq_0 \\ q &= q_0 + ry_0 \end{aligned}$$

the value r is calculated exactly and q is the correctly rounded quotient, whatever the current rounding mode.

Markstein's reciprocal theorem

The problem is that we need a perfectly rounded y_0 first, for which Markstein proves the following variant theorem.

If y_0 is within $1ulp$ of the exact $\frac{1}{b}$, then if we execute the following `fma` operations in round-to-nearest mode:

$$\begin{aligned} e &= 1 - by_0 \\ y &= y_0 + ey_0 \end{aligned}$$

then e is calculated exactly and y is the correctly rounded reciprocal, except possibly when the mantissa of b is all 1s.

Using the theorems

Using these two theorems together, we can obtain an IEEE-correct division algorithm as follows:

- Calculate approximations y_0 and q_0 accurate to 1 ulp (straightforward). [N `fma` latencies]
- Refine y_0 to a perfectly rounded y_1 by two `fma` operations, and in parallel calculate the remainder $r = a - bq_0$. [2 `fma` latencies]
- Obtain the final quotient by $q = q_0 + ry_0$. [1 `fma` latency].

There remains the task of ensuring that the algorithm works correctly in the special case where b has a mantissa consisting of all $1s$.

Can prove it will still work provided q_0 *overestimates* the true quotient.

Initial algorithm example

Uses single precision throughout:

1.	$y_0 = \frac{1}{b}(1+\epsilon)$	[`frcpa`]
2.	$e_0 = 1 - by_0$	
3.	$y_1 = y_0 + e_0y_0$	
4.	$e_1 = 1 - by_1$	$q_0 = ay_0$
5.	$y_2 = y_1 + e_1y_1$	$r_0 = a - bq_0$
6.	$e_2 = 1 - by_2$	$q_1 = q_0 + r_0y_2$
7.	$y_3 = y_2 + e_2y_2$	$r_1 = a - bq_1$
8.	$q = q_1 + r_1y_3$	

Needs 8 times the basic `fma` latency, i.e. $8 \times 5 = 40$ cycles.

Improved theorems

In proving Markstein's theorems formally in HOL, we noticed a way to strengthen them. For the main theorem, instead of requiring y_0 to be perfectly rounded, we can require only a relative error:

$$|y_0 - \frac{1}{b}| < |\frac{1}{b}|/2^p$$

where p is the floating point precision. Actually Markstein's original proof only relied on this property, but merely used it as an intermediate consequence of perfect rounding.

The altered precondition looks only trivially different, and in the worst case it is. However it is in general much easier to achieve.

Achieving the relative error bound

Suppose y_0 results from rounding a value y_0^*.

The rounding can contribute as much as $\frac{1}{2}\ ulp(y_0^*)$, which in all significant cases is the same as $\frac{1}{2}\ ulp(\frac{1}{b})$.

Thus the relative error condition after rounding is achieved provided y_0^* is in error by no more than

$$|\frac{1}{b}|/2^p - \frac{1}{2}\ ulp(\frac{1}{b})$$

Sometimes this can be significantly different.

Thus we can generalize the way Markstein's reciprocal theorem isolates a single special case.

Stronger reciprocal theorem

We have the following generalization: if y_0 results from rounding a value y_0^* with relative error better than $\frac{d}{2^{2p}}$:

$$|y_0^* - \frac{1}{b}| \leq \frac{d}{2^{2p}}|\frac{1}{b}|$$

then y_0 meets the relative error condition for the main theorem, *except possibly when the mantissa of b is one of the d largest*, i.e. when considered as an integer is $2^p - d \leq m \leq 2^p - 1$.

Hence, we can compute y_0 more ‘sloppily’, and hence perhaps more efficiently, at the cost of explicitly checking more special cases.

An improved algorithm

The following algorithm can be justified by applying the theorem with $d = 165$, explicitly checking 165 special cases.

1. $y_0 = \frac{1}{b}(1+\epsilon)$ $[\texttt{frcpa}]$
2. $d = 1 - by_0$ $q_0 = ay_0$
3. $y_1 = y_0 + dy_0$ $r_0 = a - bq_0$
4. $e = 1 - by_1$ $y_2 = y_0 + dy_1$ $q_1 = q_0 + r_0y_1$
5. $y_3 = y_1 + ey_2$ $r_1 = a - bq_1$
6. $q = q_1 + r_1y_3$

Potentially runs in 6 FP latencies, modulo parallelism/pipelining.

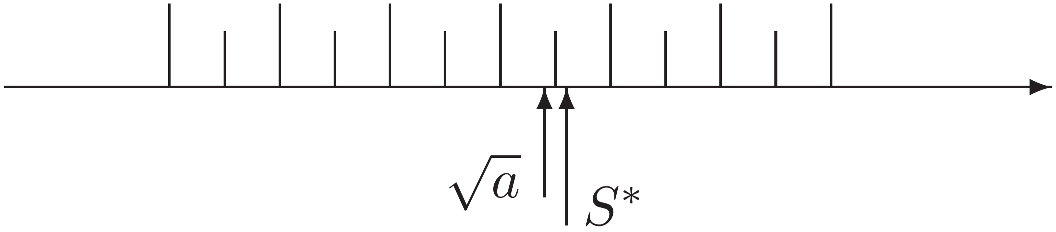
A square root algorithm

Also based on refinement of an initial approximation.

1. $y_0 = \frac{1}{\sqrt{a}}(1+\epsilon) \qquad b = \frac{1}{2}a$
2. $z_0 = y_0^2 \qquad S_0 = ay_0$
3. $d = \frac{1}{2} - bz_0 \qquad k = ay_0 - S_0 \qquad H_0 = \frac{1}{2}y_0$
4. $e = 1 + \frac{3}{2}d \qquad T_0 = dS_0 + k$
5. $S_1 = S_0 + eT_0 \qquad c = 1 + de$
6. $d_1 = a - S_1S_1 \qquad H_1 = cH_0$
7. $S = S_1 + d_1H_1$

Condition for perfect rounding

Recall the general condition for perfect rounding. We want to ensure that the two real numbers $\sqrt{a}$ and S^* never fall on opposite sides of a midpoint between two floating point numbers, as here:



Rather than analyzing the rounding of the final approximation explicitly, we can just appeal to general properties of the square root function.

Exclusion zones

It would suffice if we knew for any midpoint m that:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* cannot lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m IN midpoints fmt
        ⇒ abs(x - y) < abs(x - m))
   ⇒ round fmt Nearest x =
     round fmt Nearest y
```

Square root exclusion zones

This is possible to prove, because in fact every midpoint m is surrounded by an ‘exclusion zone’ of width $\delta_m > 0$ within which the square root of a floating point number cannot occur.

However, this δ can be quite small, considered as a relative error. If the floating point format has precision p, then we can have $\delta_m \approx |m|/2^{2p+2}$.

Example: square root of significand that’s all 1s.

Difficult cases

So to ensure the equal rounding property, we need to make the final approximation before the last rounding accurate to *more than twice* the final accuracy.

The fused multiply-add can help us to achieve *just under twice* the accuracy, but to do better is slow and complicated. How can we bridge the gap?

We can use a technique due to Marius Cornea.

Mixed analytic-combinatorial proofs

Only a fairly small number of possible inputs a can come closer than say $2^{-(2p-1)}$.

For all the other inputs, a straightforward relative error calculation (which in HOL we have largely automated) yields the result.

We can then use number-theoretic reasoning to isolate the additional cases we need to consider, then simply *try them and see*!

More than likely we will be lucky, since all the error bounds are worst cases and even if the error is exceeded, it might be in the right direction to ensure perfect rounding anyway.

Isolating difficult cases

Straightforward to show that the difficult cases have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for d a small integer.

$$2^{p+2}m = k^2 + d$$

or

$$2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen d. For example, we might be interested in whether:

$$2^{p+1}m = k^2 - 7$$

has a solution. If so, the possible value(s) of m are added to the set of difficult cases.

Solving the equations

It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$(2^{p+1} m = k^2 + d) \Rightarrow m = n_1 \vee \ldots \vee m = n_i$$

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called 'Hensel lifting'). For example, if

$$2^{25} m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and get the derived equation:

$$2^{24} m = 2k'^2 + 2k' - 3$$

By more even/odd reasoning, this has no solutions. Always recurse down to an equation that is unsatisfiable or immediately solvable.

General analytical-combinatorial setup

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.

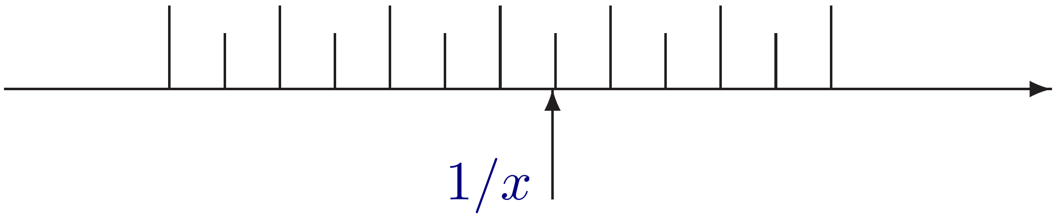
This is particularly useful given that error bounds derived from the $1 + \epsilon$ property are highly conservative.

Another 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.

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.

Typical results

```
0xFFFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFFFD 0xFE421D63446A3B34 0xFBFC17DFE0BEFF04 0xFB940B119826E598
0xFB0089D7241D10FC 0xFA0BF7D05FBE82FC 0xF912590F016D6D04 0xF774DD7F912E1F54 0xF7444DFBF7B20EAC
0xF39EB657E24734AC 0xF36EE790DE069D54 0xF286AD7943D79434 0xEDF09CCC53942014 0xEC4B058D0F7155BC
0xEC1CA6DB6D7BD444 0xE775FF856986AE74 0xE5CB972E5CB972E4 0xE58469F0234F72C4 0xE511C4648E2332C4
0xE3FC771FE3B8FF1C 0xE318DE3C8E6370E4 0xE23B9711DCB88EE4 0xE159BE4A8763011C 0xDF738B7CF7F482E4
0xDEE256F712B7B894 0xDEE24908EDB7B894 0xDE86505A77F81B25 0xDE03D5F96C8A976C 0xDDFF059997C451E5
0xDB73060F0C3B6170 0xDB6DB6DB6DB6DB6C 0xDB6DA92492B6DB6C 0xDA92B6A4ADA92B6C 0xD9986492DD18DB7C
0xD72F32D1C0CC4094 0xD6329033D6329033 0xD5A004AE261AB3DC 0xD4D43A30F2645D7C 0xD33131D2408C6084
0xD23F53B88EADABB4 0xCCCE6669999CCCD0 0xCCCE666666633330 0xCCCCCCCCCCCCCCD0 0xCBC489A1DBB2F124
0xCB21076817350724 0xCAF92AC7A6F19EDC 0xC9A8364D41B26A0C 0xC687D6343EB1A1F4 0xC54EDD8E76EC6764
0xC4EC4EC362762764 0xC3FCF61FE7B0FF3C 0xC3FCE9E018B0FF3C 0xC344F8A627C53D74 0xC27B1613D8B09EC4
0xC27B09EC27B09EC4 0xC07756F170EAFBEC 0xBDF3CD1B9E68E8D4 0xBD5EAF57ABD5EAF4 0xBCA1AF286BCA1AF4
0xB9B501C68DD6D90C 0xB880B72F050B57FC 0xB85C824924643204 0xB7C8928A28749804 0xB7A481C71C43DDFC
0xB7938C6947D97303 0xB38A7755BB835F24 0xB152958A94AC54A4 0xAFF5757FABABFD5C 0xAF4D99ADFEFCAAFC
0xAF2B32F270835F04 0xAE235074CF5BAE64 0xAE0866F90799F954 0xADCC548E46756E64 0xAD5AB56AD5AB56AC
0xAD5AAA952AAB56AC 0xAB55AAD56AB55AAC 0xAAAAB55555AAAAAC 0xAAAAAAAAAAAAAAAC 0xAAAAA00000555554
0xA93CFF3E629F347D 0xA80555402AAA0154 0xA8054ABFD5AA0154 0xA7F94913CA4893D4 0xA62E84F95819C3BC
0xA5889F09A0152C44 0xA4E75446CA6A1A44 0xA442B4F8DCDEF5BC 0xA27E096B503396EE 0x9E9B8FFFFFD8591C
0x9E9B8B0B23A7A6E4 0x9E7C6B0C1CA79F1C 0x9DFC78A4EEEE4DCB 0x9C15954988E121AB 0x9A585968B4F4D2C4
0x99D0C486A0FAD481 0x99B831EEE01FB16C 0x990C8B8926172254 0x990825E0CD75297C 0x989E556CADAC2D7F
0x97DAD92107E19484 0x9756156041DBBA94 0x95C4C0A72F501BDC 0x94E1AE991B4B4EB4 0x949DE0B0664FD224
0x942755353AA9A094 0x9349AE0703CB65B4 0x92B6A4ADA92B6A4C 0x9101187A01C04E4C 0x907056B6E018E1B4
0x8F808E79E77A99C4 0x8F64655555317C3C 0x8E988B8B3BA3A624 0x8E05E117D9E786D5 0x8BEB067D130382A4
0x8B679E2B7FB0532C 0x887C8B2B1F1081C4 0x8858CCDCA9E0F6C4 0x881BB1CAB40AE884 0x87715550DCDE29E4
0x875BDE4FE977C1EC 0x86F71861FDF38714 0x85DBEE9FB93EA864 0x8542A9A4D2ABD5EC 0x8542A150A8542A14
0x84BDA12F684BDA14 0x83AB6A090756D410 0x83AB6A06F8A92BF0 0x83A7B5D13DAE81B4 0x8365F2672F9341B4
0x8331C0CFE9341614 0x82A5F5692FAB4154 0x8140A05028140A04 0x8042251A9D6EF7FC
```

Transcendental functions: 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 $a \leq x \leq b$ 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.

```
|- (∀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.

Success and failure

HOL Light’s extensive mathematical infrastructure and complete programmability make it ideally suited for such applications.

In the hands of a skilled user — for example its author — it can be very productive. But it’s not easy for beginners:

- User is confronted with a full (and probably unfamiliar) programming language.
- Many inference rules and pre-proved theorems available, and it takes a long time to learn how to use them all.

Summary

- We need general theorem proving for some applications; it can be based on first order set theory or higher-order logic.
- In practice, we need a combination of interaction and automation for difficult proofs.
- LCF gives a good way of realizing a combination of soundness and extensibility.
- Different proof styles may be preferable, and they can be supported on top of an LCF-style core.