Formal Verification In Industry (II)

John Harrison

Intel Corporation

- Introduction
- Quick overview of LCF and HOL.
- Floating point numbers and IA-64 formats
- HOL floating point theory
- Square root algorithm
- Correctness proof in HOL

Introduction

We have seen that equivalence checking, model checking, STE etc. are powerful tools for verification, and are not limited to hardware. However, for higher-level verification, e.g. against specifications involving real numbers, a more general mathematical framework is required.

A good example is the verification of certain floating point operations. For this purpose, one needs a more powerful theorem prover that is capable of reasoning about the underlying mathematics. One prover that is being applied inside Intel for this purpose is HOL Light.

In this talk we will show how HOL Light has been used to formalize floating point arithmetic and applied to formal verification of mathematical algorithms. We will use a square root software algorithm as an example; it is implemented using basic IA-64 instructions.

The LCF approach

The prover we use, HOL Light, is based on the approach to theorem proving pioneered in Edinburgh LCF in the 70s. The key ideas are:

- All theorems created by low-level primitive rules.
- Guaranteed by using an abstract type of theorems; no need to store proofs.
- ML available for implementing derived rules by arbitrary programming.

This gives advantages of reliability and extensibility. The system's source code can be completely open. The user controls the means of production (of theorems). To improve efficiency one can:

- Encapsulate reasoning in single theorems.
- Separate proof search and proof checking.

Some primitive rules of HOL Light

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

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

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

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

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

There are a few more similar primitive rules, and two rules of definition.

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.

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

Floating point numbers

There are various different schemes for floating point numbers. Usually, the floating point numbers are those representable in some number n of significant binary digits, within a certain exponent range, i.e.

$$(-1)^s \times d_0.d_1 d_2 \cdots d_n \times 2^e$$

where

- $s \in \{0, 1\}$ is the *sign*
- $d_0.d_1 d_2 \cdots d_n$ is the *significand* and $d_1 d_2 \cdots d_n$ is the *fraction*. These are not always used consistently; sometimes 'mantissa' is used for one or the other
- e is the exponent.

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

IA-64 floating point formats

A floating point format is a particular allowable precision and exponent range.

IA-64 supports a multitude of possible formats, e.g.

- IEEE single: $p = 24$ and $-126 \leq e \leq 127$
- IEEE double: $p = 53$ and $-1023 \leq e \leq 1023$
- IEEE double-extended: $p = 64$ and $-16382 \leq e \leq 16383$
- IA-64 register format: $p = 64$ and $-65534 \leq e \leq 65535$

There are various other hybrid formats, and a separate type of parallel FP numbers, which is SIMD single precision.

The highest precision, 'register', is normally used for intermediate calculations in algorithms.

HOL floating point theory (1)

We have formalized a generic floating point theory in HOL, which can be applied to all the IA-64 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`.

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. that a real number rounds to itself if it is in the floating point format:

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

that rounding is monotonic:

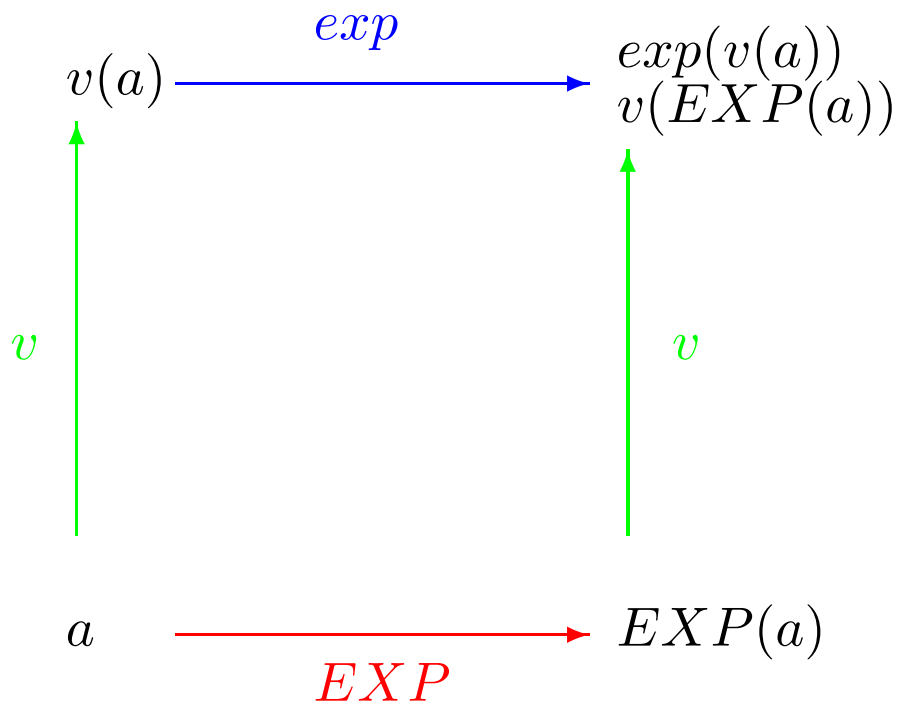
```
|- ¬(precision fmt = 0) ∧ x <= y
   ⟹ round fmt rc x <= round fmt rc y
```

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
```

Floating point correctness (1)

Floating point operations are considered correct when they yield an answer that is sufficiently close to the exact mathematical answer. For example, for the exponential function:



In general, we can't represent the exact mathematical result as a floating point number.

Floating point correctness (2)

The IEEE standard states that all the algebraic operations should give the closest floating point number to the true answer, or the closest number up, down, or towards zero in other rounding modes.

However, the standard makes no recommendations about the transcendentals. It is difficult to guarantee such a stringent criterion, because of the 'table maker's dilemma'.

Being able to approximate a real number arbitrarily closely does *not* in general mean that one can decide the correctly rounded digits in a positional expansion.

$$e^{1.626} = 5.083499996273\ldots$$

To decide whether 5.0834 or 5.0835 is the closest 5-digit number, we need to perform the evaluation to much more than an accuracy of 10^{-5}.

Our algorithm example

Our example is an algorithm for square roots using only single precision computations (hence suitable for SIMD). It is built using two basic IA-64 operations:

- The reciprocal square root approximation `frsqrta`, which given an input a returns an approximation to $1/\sqrt{a}$ with relative error at most about $2^{-8.85}$.
- The fused multiply add and its negated variant, which calculates $xy + z$ or $z - xy$ with just a single rounding error.

Because it only uses single precision calculations, readers can 'try it at home'; it's fairly easy to simulate a single-precision fused multiply-add on standard hardware. The actual tables used in the `frcpa` instruction are documented in the IA-64 Architecture Guide.

`http://developer.intel.com/design/ia64/downloads/adag.htm`

The square root algorithm

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

Proving IEEE correctness

Provided the input number is in a certain range, this algorithm returns the correctly rounded square root *and* sets all the IEEE flags correctly.

How do we prove that the result is correctly rounded? We will concentrate on round-to-nearest mode, which is the most interesting case. What the algorithm actually returns is the result of rounding the value:

$$S^* = S_1 + d_1 H_1$$

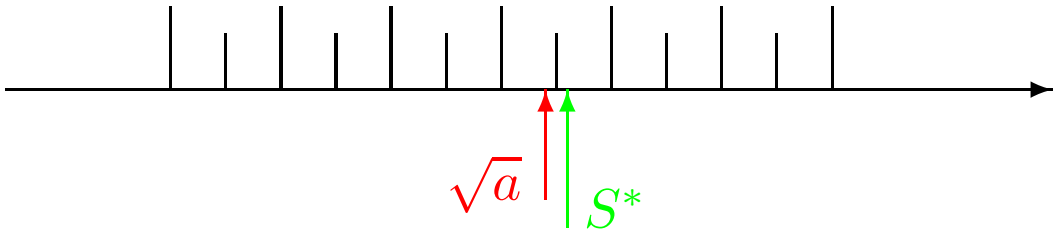
The algorithm is correct if this is always the same as the result of rounding the exact square root $\sqrt{a}$.

Moreover, properties of this value S^*, e.g. whether it is already exactly a floating point number, determine the final flag settings (intermediate steps do not set flags). We also want to make sure these properties are the same as for the exact square root.

Condition for perfect rounding

A sufficient condition for perfect rounding is that the closest floating point number to $\sqrt{a}$ is also the closest to S^*. That is, the two real numbers $\sqrt{a}$ and S^* never fall on opposite sides of a midpoint between two floating point numbers.

In the following diagram this is not true; $\sqrt{a}$ would round to the number below it, but S^* to the number above it.



How can we prove this?

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)
```

And 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 $\delta_m \approx |m|/2^{2p+2}$.

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?

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

By some straightforward mathematics, formalizable in HOL without difficulty, one can 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. (Typically ≤ 10, depending on the exact accuracy of the final approximation before rounding.)

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

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:

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

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. In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable. One equation can split into two, but never more.

Conclusions

Because of HOL's mathematical generality, all the reasoning needed can be done in a unified way with the customary HOL guarantee of soundness:

- Underlying pure mathematics
- Formalization of floating point operations
- Proof that the condition tested ensures perfect rounding
- Routine relative error computation for the final result before rounding
- Number-theoretic isolation of difficult cases
- Explicit computation with those cases

Moreover, because HOL is programmable, many of these parts can be, and have been, automated. In short, HOL is an almost ideal vehicle for verifications of this type.