# Floating point verification: the exponential function

John Harrison

University of Cambridge

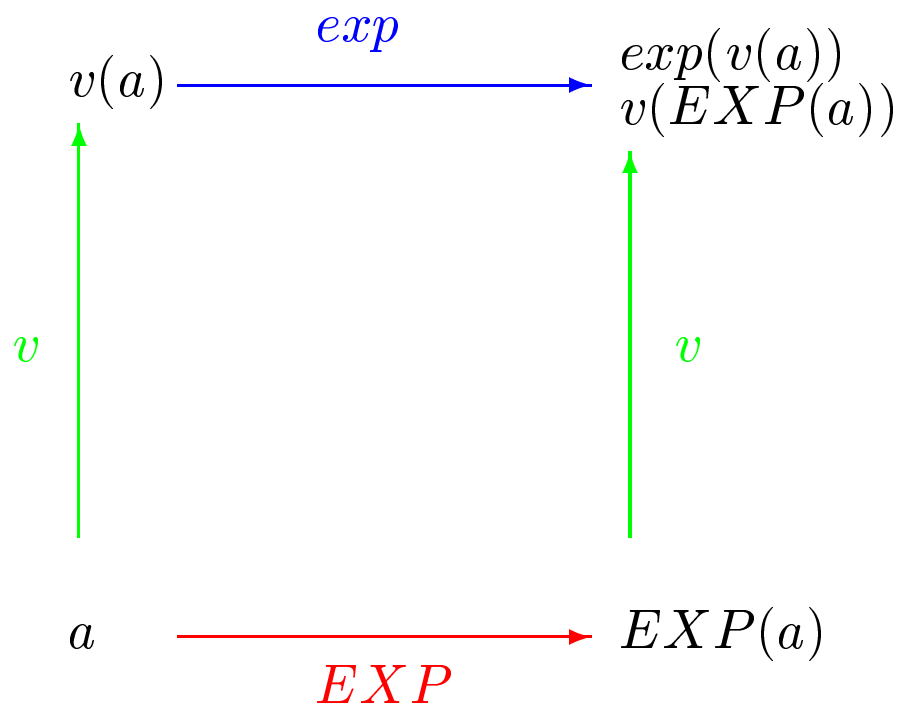
- General introduction
- The correctness statement we want
- Our implementation language
- Sketch of the algorithm
- Organization of HOL proof
- Formalizing IEEE arithmetic
- Details of the proof
- Conclusions

# General introduction

- Floating point algorithms are fairly small, but often complicated mathematically.
- There have been errors in commercial systems, e.g. the Pentium FDIV bug in 1994.
- In the case of transcendental functions it's difficult even to say what correctness means.
- Verification using model checkers is difficult because of the need for mathematical apparatus.
- It can even be difficult using theorem provers since not many of them have good theories of real numbers etc.

# Floating point correctness

We want to specify the correctness according to the following diagram:




We measure the difference between $v(EXP(a))$ and $exp(v(a))$ in 'units in the last place' of $EXP(a)$.

# Units in the last place

In fact, the notion of a 'unit in the last place' is used in two slightly different ways, as the value of the least significant bit of the binary interval containing:

- The rounded result (e.g. Goldberg's paper 'What every computer scientist should know about floating point arithmetic').
- The exact result (e.g. Muller, 'Elementary Functions').

In the proof here, we use the first measure, since this is clearly what is intended by the author of the paper we were following. Besides, it makes no difference in this case.

Tang doesn't mention this issue at all, though in his later paper on the logarithm he does mention this 'subtlety'. Getting a clear specification is one of the intended benefits of formalization.

# Our implementation language

This includes the following constructs:

*command* = `variable :=` *expression*
| *command* `;` *command*
| `if` *expression* `then` *command*
`else` *command*
| `if` *expression* `then` *command*
| `while` *expression* `do` *command*
| `do` *command* `while` *expression*
| `skip`
| { *expression*}

We have a simple relational semantics in HOL, and derive weakest preconditions and total correctness rules. We then prove total correctness via VC generation.

The idea is that this language can be formally linked to C, Verilog, Handel, ...

# Sketch of the algorithm

The algorithm we verify is taken from a paper by Tang in *ACM Transactions on Mathematical Software*, 1989.

Similar techniques are widely used for floating point libraries, and, probably, for hardware implementations.

The algorithm relies on a table of precomputed constants. Tang's paper gives actual values as hex representations of IEEE numbers.

We can split the operations into three steps:

- Perform range reduction
- Use polynomial approximation
- Reconstruct answer using tables

# Mathematics of the algorithm

First we obtain a reduced argument $r$ such that for some integer $n$:

$$x = n\frac{ln(2)}{32} + r$$

and $-\frac{ln(2)}{64} \leq r \leq \frac{ln(2)}{64}$. This $n$ is found by rounding $x\frac{32}{ln(2)}$ to the nearest integer. Now we decompose $n$ into its quotient and remainder when divided by 32, i.e. $n = 32m + j$ with $0 \leq j \leq 31$. Hence

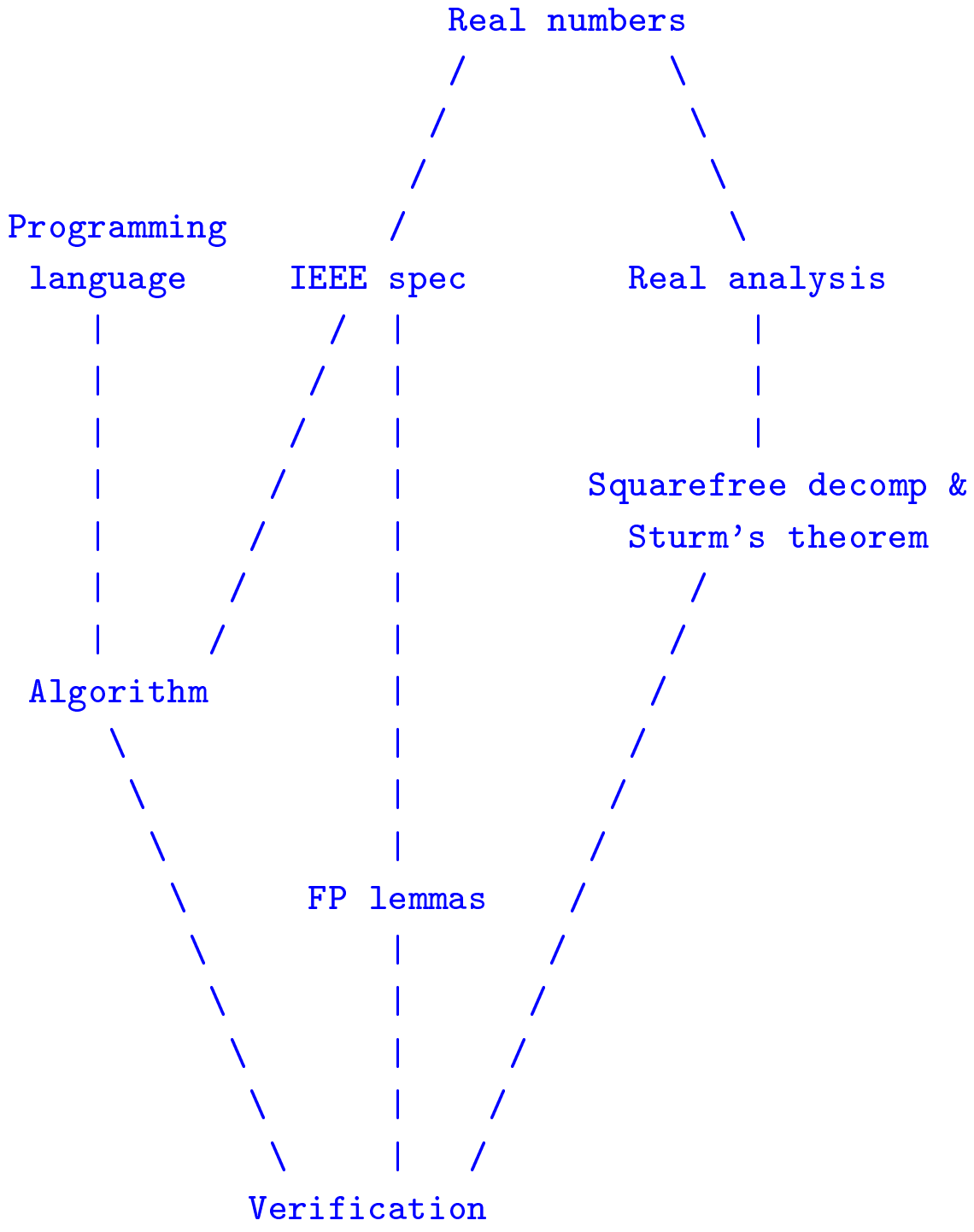
$$e^x = e^{(32m+j)\frac{ln(2)}{32}+r} = e^{ln(2)m}e^{\frac{ln(2)j}{32}}e^r = 2^m 2^{\frac{j}{32}} e^r$$

Values of $2^{\frac{j}{32}}$ for $0 \leq j \leq 31$ are prestored constants, and multiplication by $2^m$ is fast. Hence we just need to calculate $e^r$ for $r \in [-\frac{ln(2)}{64}, \frac{ln(2)}{64}]$. This is done by a low-order polynomial approximation $p(r) \approx e^r - 1$.

# Code for the algorithm

```
if Isnan(X) then E := X
else if X == Plus_infinity then E := Plus_infinity
else if X == Minus_infinity then E := Plus_zero
else if abs(X) > THRESHOLD_1 then
  if X > Plus_zero then E := Plus_infinity
  else E := Plus_zero
else if abs(X) < THRESHOLD_2 then E := Plus_one + X
else
 (N := INTRND(X * Inv_L);
  N2 := N % Int_32;
  N1 := N - N2;
  if abs(N) >= Int_2e9 then
    R1 := (X - Tofloat(N1) * L1) - Tofloat(N2) * L1
  else
    R1 := X - Tofloat(N) * L1;
  R2 := Tofloat(--N) * L2;
  M := N1 / Int_32;
  J := N2;
  R := R1 + R2;
  Q := R * R * (A1 + R * A2);
  P := R1 + (R2 + Q);
  S := S_Lead(J) + S_Trail(J);
  E1 := S_Lead(J) + (S_Trail(J) + S * P);
  E := Scalb(E1,M)
 )
```

# Organization of HOL proof



# Formalizing IEEE arithmetic

We formalize the main parts of the IEEE-754 standard in HOL. This is tedious but straightforward; it's all been done before in Z and PVS.

- Floating point formats
- Categorization of numbers (zero, denormal, normal, infinity, NaN)
- Bit encodings and real number valuation
- Directed roundings to reals and integers
- Operations, including exceptional arguments

We do not consider the distinction between quiet and signalling NaNs, exception generation and trap handlers. This would be difficult without specifying the ambient computing environment.

# Floating point lemmas (1)

We define the error resulting from rounding a real number to a floating point value:

```
|- error(x) =
   Val(float(round(float_format) To_nearest x)) - x
```

Because of the regular way in which the operations are defined, all the operations then relate to their abstract mathematical counterparts according to the same pattern:

```
|- Finite(a) ∧ Finite(b) ∧
   abs(Val(a) + Val(b)) < threshold(float_format)
   ⟹ Finite(a + b) ∧
       (Val(a + b) = (Val(a) + Val(b)) +
       error(Val(a) + Val(b)))
```

The comparisons are even more straightforward:

```
|- Finite(a) ∧ Finite(b)
   ⟹ (a < b = Val(a) < Val(b))
```

# Floating point lemmas (2)

We have several lemmas quantifying the error, of which the most useful is the following:

```
|- abs(x) < threshold(float_format) ∧
   abs(x) < (&2 pow j / &2 pow 125)
   ⟹ abs(error(x)) <= &2 pow j / &2 pow 150
```

There are many important situations, however, where the operations are exact, because the result is exactly representable. Trivially, for example, the negation and absolute value functions are always exact:

```
|- Finite(a)
   ⟹ Finite(abs(a)) ∧ (Val(abs(a)) = abs(Val(a)))
```

Also, if a result only has 24 significant digits (modulo some care in the denormal case), then it is exactly representable:

```
|- (abs(x) = (&2 pow e / &2 pow 149) * &k) ∧
   k < 2 EXP 24 ∧ e < 254
   ⟹ ∃a. Finite(a) ∧ (Val(a) = x)
```

# Floating point lemmas (3)

Any calculation whose result is exactly representable has an error of zero:

```
|- Finite(a) ∧ Finite(b) ∧
   Finite(c) ∧ (Val(c) = Val(a) * Val(b))
   ⟹ Finite(a * b) ∧
       (Val(a * b) = Val(a) * Val(b))
```

Another important case of exact operations is subtraction of nearby values with the same sign:

```
|- Finite(a) ∧ Finite(b) ∧
   &2 * abs(Val(a) - Val(b)) <= abs(Val(a))
   ⟹ Finite(a - b) ∧
       (Val(a - b) = Val(a) - Val(b))
```

This is a classic result in floating point error analysis.

We also have a type of machine integers, and prove various obvious results about how the arithmetic operations on those work.

# Error in range reduction

This part is quite difficult, as the code is very delicately written to ensure that $R_1$ is calculated exactly. The stored values $L_1$ and $L_2$ have enough trailing zeros that multiplication by small enough integers is exact; this is a fairly straightforward application of earlier lemmas.

More difficult is establishing that the subsequent subtractions are exact by virtue of cancellation. We can't quite use the previous lemmas and we end up using this ad hoc lemma, which says that subtraction of $NL_1$ from any value within $\frac{1}{88}$ of it is exact.

```
|- (L1 = float (0,(121,3240448))) ∧
   Finite(X) ∧
   Finite(Tofloat(N) * L1) ∧
   (Val(Tofloat(N) * L1) = Ival(N) * Val(L1)) ∧
   abs(Val(X) - Val(Tofloat(N) * L1)) <= inv(&88)
   ⟹ Finite(X - Tofloat(N) * L1) ∧
      (Val(X - Tofloat(N) * L1) =
       Val(X) - Val(Tofloat(N) * L1))
```

# Error in polynomial approximation

This part is also tricky. In brief, these are the steps:

- Prove that the error in a high-order Taylor series is much better than we need.
- Consider the difference between this and the minimax polynomial actually used.
- Locate the zeros of (the squarefree decomposition of) its derivative.
- Prove using Sturm's theorem that these are all the zeros.
- Hence get a bound on the error by evaluation at the endpoints of the interval and the points of zero derivative, using some elementary real analysis.

Tang makes a small slip over the necessary interval.

# Error in reconstruction

This consists of adding up rounding errors together with the fact that the table entries for $2^{\frac{j}{32}}$ are not exact (these errors are much smaller than the rounding errors, though).

We exploit HOL's programmability to add/multiply and bound the dozens of error terms automatically.

Although we make no simplifying assumptions, as Tang does, we actually end up with a sharper error.

Tang derives bounds of $0.5267ulp$ and $0.5378ulp$ in $E1$, depending on the binary interval in which it lies, $[\frac{1}{2}, 1)$ or $[1, 2)$. Our bounds are $0.5125ulp$ and $0.5338ulp$ respectively.

The better bound for the second results purely from HOL's mechanical application of the theorems about error bounds.

# Overflow and underflow

We also prove that the real and approximate exponentials have the same overflow behaviour. If there is a disparity in the overflow behaviours, then $2^M Val(E1)$ and $e^{Val(X)}$ lie on opposite sides of the overflow threshold.

This means that $2^M Val(E1)$ is at least as close to the overflow threshold as it is to $e^{Val(X)}$, that is, within about $0.54\ 2^M/2^{23}$. Thus $|threshold/e^{Val(X)} - 1| < 0.55/2^{22}$. Hence by appealing to the following theorem:

```
|- abs(x - &1) <= e ∧ e <= inv(&4)
   ⟹ abs(ln(x)) <= e + e pow 2
```

we find that $|ln(threshold) - Val(X)| \leq 2^{-22}$. However we can prove this is impossible because $ln(threshold)$ is further away than that from any representable number (the closest is `float(0,133,3240472)`).

There are similar subtleties when the result just denormalizes.

# Checking of prestored constants

The final correctness result is given under the assumption that various constants have the hex values Tang says they do. From these, we need to derive mathematical properties.

Many of them are related to $ln(2)$, so we can get all those from a highly accurate rational approximation to $ln(2)$ obtained from Taylor's theorem:

```
|- abs(ln(&2) - &54453198020265458334082568662084 7 /
                &78559358744381708183222972579840 0)
   < inv(&2 pow 51)
```

We can justify the table entries for $2^{\frac{j}{32}}$ using the following theorem, easily derived in HOL from the Mean Value Theorem for derivatives:

```
|- &0 < a ∧ a <= &2 ∧
   &0 < b ∧ b <= &2 ∧
   abs(b pow 32 - a pow 32) <= e
   ⟹ &32 * (a pow 32 - e) * abs(b - a) <= &2 * e
```

# The final result

Under the various 'definitional' assumptions, we confirm Tang's bottom-line result:

```
(Isnan(X) ⟹ Isnan(E)) ∧
(X == Plus_infinity ∨
 Finite(X) ∧
 exp(Val X) >= threshold(float_format)
 ⟹ E == Plus_infinity) ∧
(X == Minus_infinity ⟹ E == Plus_zero) ∧
(Finite(X) ∧ exp(Val X) < threshold(float_format)
 ⟹ Isnormal(E) ∧
     abs(Val(E) - exp(Val X))
     < (&54 / &100) * Ulp(E) ∨
     (Isdenormal(E) ∨ Iszero(E)) ∧
     abs(Val(E) - exp(Val X))
     < (&77 / &100) * Ulp(E))
```

This is somewhat more explicit than Tang's statement regarding overflow.

# Conclusions

- We confirm (and strengthen) the main results of the hand proof. But we detect a few slips and uncover subtle issues. This class of proofs is a good target for verification.
- The proof was very long (over 3 months of work), but most of this was devoted to general results that could be re-used.
- The necessary infrastructure for the proof exists only in HOL Light; the most plausible alternative would be PVS.
- Automation of linear arithmetic is practically indispensable. Better tools for nonlinear reasoning are needed.
- The proof runtimes are very long owing to the extensive use of arithmetic done by inference. Some support for the ACL2 worldview.