Floating point verification in HOL Light: the exponential function

John Harrison

University of Cambridge

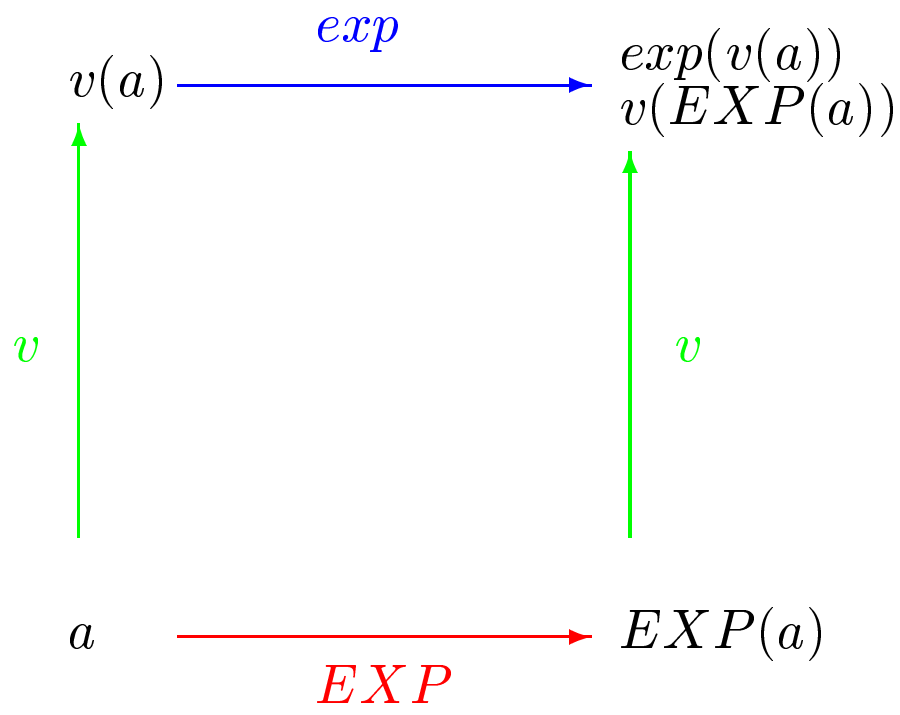
- Introduction
- Floating point correctness
- Our implementation language
- The algorithm
- Outline of the HOL proof
- General conclusions

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 $EXP(a)$.

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 define 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, ...

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.

The algorithm works in three phases:

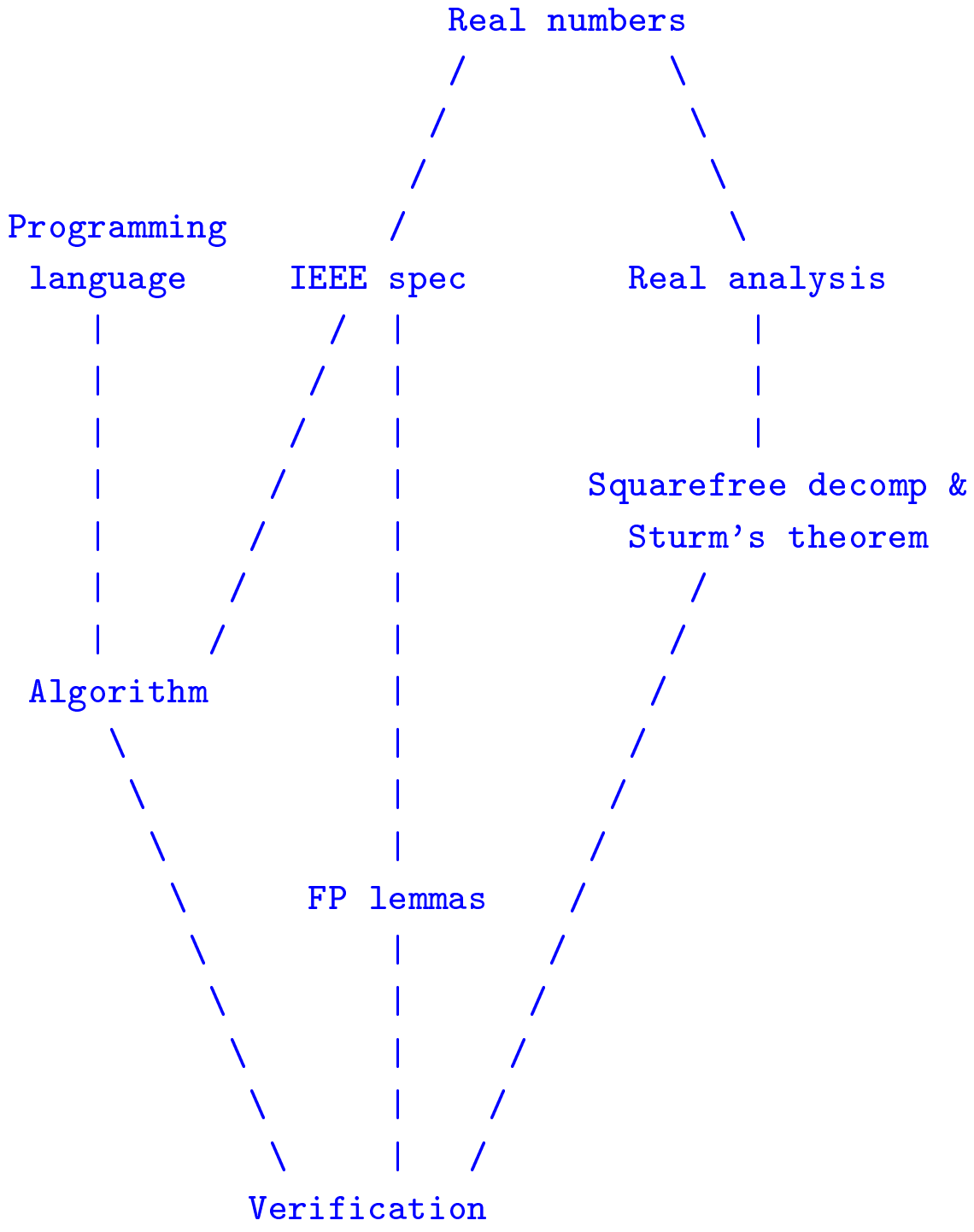
- Perform range reduction
- Use polynomial approximation
- Reconstruct answer using tables

The correctness proof reflects this.

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)
 )
```

Structure of the HOL proof



Floating point lemmas (1)

We define the error `error(x)` resulting from rounding a real number `x` to a floating point value.

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, e.g.

```
|- 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, e.g. 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.

Informal error analysis

Tang's error analysis translates quite directly into HOL. One needs to:

1. Prove that clever implementation tricks ensure certain remainder terms are calculated exactly. This relies on cancellation, and the fact that pre-stored constants have trailing zeroes.

2. Prove that the polynomial approximation obeys the appropriate error bounds.

3. Prove that the rounding errors when reconstructing the final answer do not get too large.

In Tang's paper, 1 is quite brief, 2 is dismissed in a few lines, while 3 is given a long and detailed proof.

HOL error analysis

In the HOL version, this order of difficulty is reversed!

1. The first part is not fundamentally difficult, but quite tricky because it involves a lot of special cases and low-level proofs.

2. The second part involves numerical approximation, which needs a lot of work to translate into a formal proof (e.g. Taylor series, Sturm's theorem ...). In fact Tang makes a small mistake here, though it doesn't affect the final result.

3. The last part is quite routine, and we can program HOL to compose the rounding errors automatically. Actually, we derive better bounds than Tang does since we avoid making simplifying assumptions to cut down the work.

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))
```

In fact, this specification is a bit more precise than Tang's, e.g. we are explicit about the overflow threshold.

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.
- It's a mistake to believe that only 'trivial' mathematics is needed for verification applications. HOL Light's mathematical theories are essential.
- 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.