Floating point verification in HOL Light: the exponential function

John Harrison

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England

Abstract. Since they often embody compact but mathematically sophisticated algorithms, operations for computing the common transcendental functions in floating point arithmetic seem good targets for formal verification using a mechanical theorem prover. We discuss some of the general issues that arise in verifications of this class, and then present a machine-checked verification of an algorithm for computing the exponential function in IEEE-754 standard binary floating point arithmetic. We confirm (indeed strengthen) the main result of a previously published error analysis, though we uncover a minor error in the hand proof and are forced to confront several subtle issues that might easily be overlooked informally.

1 Introduction

Algorithms for performing floating point operations are often rather complex. It is difficult to make sure they are correct — for example, a bug in the floating-point division instruction of the Intel Pentium gained widespread publicity quite recently (Pratt 1995). We present here a machine-checked proof of a floating point algorithm, with the hope of illustrating the potential for theorem provers in this area. The present discussion is necessarily sketchy, and we refer the reader to an earlier technical report (Harrison 1997a) for a more detailed and discursive treatment.

Suppose we have a set $\mathbb{F}$ of floating point numbers. For the purposes of this general discussion, we will ignore special cases such as infinities and NaNs,[1] and suppose that each member of $\mathbb{F}$ has a corresponding real number value. We will use v to denote the valuation function $\mathbb{F} \to \mathbb{R}$.

We specify correctness of the floating point operation EXP by comparing its output with the true mathematical result, using the valuation function v to mediate between the realms of floating point and real numbers. That is, for each floating point number a, we compare $exp(v(a))$ and $v(\mathrm{EXP}(a))$; in algebraist's jargon, we are interested in how closely the following diagram 'almost commutes':

[1] NaN = Not a Number, a special value to represent the result of an erroneous calculation such as taking the square root of a negative number.

$$\begin{tikzcd}
v(a) \arrow[rr, "exp"] & & \begin{array}{l} exp(v(a)) \\ v(\mathrm{EXP}(a)) \end{array} \\
a \arrow[u, "v"] \arrow[rr, "\mathrm{EXP}"'] & & \mathrm{EXP}(a) \arrow[u, "v"']
\end{tikzcd}$$

In the present verification, we quantify the relative error in the sense of 'units in the last place' (*ulp*). One *ulp* is the magnitude of the least significant bit of the value concerned. (When making formal statements we will follow our HOL formalization and regard *ulp* as a function of the relevant floating point number, writing $ulp(a)$ or ulp_a for one unit in the last place of a.) Note that perfect rounding would ensure that the error in the output was always ≤ 0.5 *ulp*, but owing to the 'table maker's dilemma' (Goldberg 1991) this is difficult for the transcendentals.

2 HOL Light

The verification described here is conducted using HOL Light (Harrison 1996a), a version of the HOL prover (Gordon and Melham 1993), itself descended from Edinburgh LCF (Gordon and Melham 1993). We cannot describe the system in detail here, and to help the reader unfamiliar with HOL, we will use standard symbols for the logical operators. Most of the operations on natural and real numbers should look familiar even in their HOL ASCII representations, but here are some of the less obvious ones together with renderings in standard mathematical notation and English:

HOL notation	Standard symbol	Meaning
`SUC(n)`	$n+1$	Successor operation on $\mathbb{N}$
`m EXP n`	m^n	Natural number exponentiation
`&`	(*none*)	Natural map $\mathbb{N} \to \mathbb{R}$
`--x`	$-x$	Unary negation of x
`inv(x)`	x^{-1}	Multiplicative inverse of x
`abs(x)`	$\|x\|$	Absolute value of x
`x pow n`	x^n	Real x raised to natural number power n
`root n x`	$\sqrt[n]{x}$	Positive n^{th} root of x
`Sum(n,d) f`	$\Sigma_{i=n}^{n+d-1} f(i)$	Sum of d terms $f(i)$ starting with $f(n)$

HOL's type system distinguishes natural numbers and reals, so `&` is used to map between them. It's mainly used as part of real number constants like `&0`, `&1`

etc. Note also that while one might prefer to regard 0^{-1} as 'undefined' (in some precise sense), we set `inv(&0) = &0` by definition.

The floating point algorithm is expressed in a simple imperative 'while language'. Following Gordon (1989), this is given a precise operational semantics inside HOL, while at the same time remaining a recognizable vehicle similar to a subset of typical programming or hardware description languages.

A HOL term '`correct` $p\ C\ q$' means that, if started in any state satisfying p, the command (program fragment) C will terminate in a state satisfying q. It is assertions of this form that we want to prove. In order to make this process easier, we use the standard technique of verification condition generation. The user presents a program together with loop annotations and possibly other assertions. HOL automatically processes this and produces a series of proof obligations. All these are purely concerned with facts in the underlying theories, and have no explicit connection with programming constructs. When they are all proved, HOL will reconstruct a proof that the program, stripped of its annotations if desired, obeys the required correctness criteria.

3 Formalizing IEEE arithmetic

We will therefore assume that our ultimate goal is to deal with IEEE-754 floating point numbers, and moreover, that where we use subsidiary floating point arithmetic operations such as addition and multiplication, they conform to the IEEE standard. Therefore, an important preliminary step in our work is to translate key parts of IEEE-754 into formal HOL specifications. We do not attempt an exhaustive exegesis of the standard — for some parts, e.g. the specification of trap handlers, this would be possible only as part of a more general specification of an ambient computing environment. We just aim to formalize enough to make it clear that any implementations we verify do in fact conform to the standard in the essential details. Our effort can be compared to a Z specification by Barratt (1989) and a PVS specification of IEEE-854 by Miner (1995). We have found the latter particularly valuable in our work; it is easy to read and the underlying logic is similar to HOL's.

The operations on floating point numbers are distinct from those on mathematical real numbers, natural numbers etc. However, HOL Light features operator overloading, so we use the conventional symbols for arithmetic operations on floats; HOL exploits types to disambiguate input, signalling an error if it is unable to do so. There is one exception: we cannot overload the equality operation as it is already polymorphic. Instead we use `==` for equality on floating point values, which should look familiar to C programmers. Arguably, using the conventional symbol for equality would be rather misleading, since the standard notion is neither reflexive ($x \neq x$ if x is a NaN) nor substitutive ($-0 = +0$ but $\frac{1}{+0} \neq \frac{1}{-0}$).

The reader may need to bear in mind this overloading in order to understand some of the HOL theorems that follow. It is usually easy to see what the type of an operation is by noting how it is combined with a function of known type such as the valuation function $Val : \mathbb{F} \to \mathbb{R}$ (this is in fact how HOL's typechecker does it). For example, in `Val(a + b)`, the addition operation must be a floating point operation, while in `Val(a) + Val(b)` the addition operator is mathematical real addition. Generally, overloading always makes some expressions more natural at the cost of requiring a certain alertness on the part of the reader. In informal explanations we may omit Val where we should properly write it.

As well as floating point numbers, the algorithm we are concerned with uses machine integers. We will not discuss the (straightforward) HOL formalization of these numbers in detail, but we will note that they are 2s complement 32-bit integers. Their behaviour on overflow is undefined, i.e. a fixed arbitrary value results. This seems the most pessimistic assumption that can reasonably be made about typical contemporary hardware. In fact, the correctness of the algorithm only requires a pretty limited range of integers.

There are coercions `Tofloat` and `Toint` for mapping between floats and ints, and these are of course subject to range restrictions in each case. We will just be explicit about two operations used later, to avoid any ambiguity. The integer modulus operation, written `%` as in C, always returns a positive answer whatever the sign of its arguments. The function `INTRND` is the composition of the round-to-integer-value operation on floating point numbers and the coercion function `Toint`. One further operation that we use, recommended but not mandated by the IEEE Standard, is `Scalb`, where `Scalb(a,N)` scales the floating point number a by 2^N for an integer N. This operation is performed atomically, so may avoid overflow even if 2^N itself would overflow.

4 Lemmas about floating point numbers

In order to perform error analysis reasonably smoothly, we try to arrive at fairly general theorems about the floating point operations. Occasionally we need to dive down to more specialized and intricate results — we will present an example later — but for the most part we can use a standard collection of lemmas. Following Tang (1989), we define the error resulting from rounding a real number x to a floating point value as `error(x)`. Now because of the regular way in which the operations are defined, they all relate to their abstract mathematical counterparts according to the same pattern for finite operands:

```
|- 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, e.g.

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

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

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
```

The situation for numbers in the denormal range is slightly worse:

```
|- ∀x. abs x < inv (&2 pow 126) ⟹ abs(error x) <= inv (&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)
```

and the error in any calculation with an exactly representable result is zero, e.g.

```
|- 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. This is a well-known result in floating point error analysis, similar to Theorem 11 of Goldberg (1991).

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

5 The algorithm

The algorithm we verify is one given by Tang (1989) for the exponential function. Tang gives a fairly detailed explanation of the algorithm together with an error analysis, which makes it a suitable target for formal treatment. It is based on a table of precomputed values with a polynomial used to interpolate between them. To achieve good accuracy, several values are split across two floating point number, one much larger than the other. Moreover, the ordering of certain operations is often critical. The following is a rendering of the algorithm in our while-language, embedded in the correctness assertion that is the ultimate conclusion of the work described here.

```
|- (Int_32          = Int(32)) ∧
   (Int_2e9         = Int(2 EXP 9)) ∧
   (Plus_one        = float(0,127,0)) ∧
   (THRESHOLD_1     = float(0,134,6056890)) ∧
   (THRESHOLD_2     = float(0,102,0)) ∧
   (Inv_L           = float(0,132,3713595)) ∧
   (L1              = float(0,121,3240448)) ∧
   (L2              = float(0,102,4177550)) ∧
   (A1              = float(0,126,68)) ∧
   (A2              = float(0,124,2796268)) ∧
   TABLES_OK S_Lead S_Trail
   ⟹
   correct
   T
   var X:float,E:float,R1:float,R2:float,R:float,P:float,Q:float,
       S:float, E1:float, N:Int,N1:Int,N2:Int,M:Int,J:Int;
     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)
      )
   end
   (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))
```

The constant TABLES_OK is used to abbreviate a large set of assumptions about the values of table entries. All the following values are taken from Tang's paper.

```
|- TABLES_OK S_Lead S_Trail =
       (S_Lead(Int 0)   = float(0,127,0)) ∧
       (S_Lead(Int 1)   = float(0,127,183680)) ∧
       ...
       (S_Lead(Int 31)  = float(0,127,8029056)) ∧
       (S_Trail(Int 0)  = float(0,0,0)) ∧
       (S_Trail(Int 1)  = float(0,106,5444997)) ∧
       ...
       (S_Trail(Int 31) = float(0,109,4943305))
```

6 The HOL verification

Following Tang, we split the error into three more or less independent parts and analyze them separately.

- The error in range reduction.
- The error in the polynomial approximation.
- The rounding error in the reconstruction of the result.

6.1 Checking of prestored constants

In several places, we need to prove mathematical results about the various prestored constants. Most of these constants bear a straightforward relationship to $ln(2)$, so first we obtain, by formal proof, an accurate approximation to $ln(2)$. We use $ln(2) = ln(1+\frac{1}{2}) - ln(1-\frac{1}{4})$, since each of the values on the right of this equation can be evaluated reasonably efficiently by truncating its Taylor series, using existing HOL theorems about the error in such situations. The numerical result is:

```
 |- abs(ln(&2) - &544531980202654583340825686620847 /
                 &785593587443817081832229725798400) < inv(&2 pow 51)
```

The other constants needed are approximations to $s_j = 2^{\frac{j}{32}}$ for $j = 0, \ldots, 31$. The easiest way to justify these is to measure the difference $s_j^{32} - 2^j$ and appeal to the following theorem, easily derived in HOL from the Mean Value Theorem for derivatives:

```
|- &0 < x ∧ x <= &2
   ⟹ abs(x - root 32 (&2 pow j))
       <= abs(x pow 32 - &2 pow j) /
          (if x pow 32 <= &2 pow j then &32 * x pow 32 / x
           else &16 * &2 pow j)
```

This allows us to deduce that the difference between the stored value s_j and the true mathematical figure $2^{\frac{j}{32}}$ is below 2^{-41} in all cases. This could be sharpened a little, but it is already much better than the accuracy we need later. There is one exception: we need later the fact that in the case $j = 0$ the error is zero, trivially so since s_0 is exactly 1.

6.2 Error in range reduction

The error analysis here is rather intricate, belying the simplicity of the code. We must establish 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 about the exact representability of values with few enough significant digits. More difficult is establishing that the subsequent subtractions, either $X - NL_1$ or both $X - N_1L_1$ and $(X - N_1L_1) - N_2L_1$ depending on the arm of the conditional, are exact by virtue of cancellation. We are tantalizingly close to being able to apply a previous lemma requiring that $2|X - NL| \leq |NL|$ or $2|X - NL| \leq |X|$. This always works for the subtraction $X - N_1L_1$, but the required preconditions are just missed in the other subtractions in the cases when $N = \pm 1$ or $N_2 = 1$ respectively. We need to analyze L_1's bit pattern more carefully to justify the exactness of subtraction over a larger range. This tedious reasoning is embedded in the following 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))
```

6.3 Error in polynomial approximation

This is a matter of pure mathematics, and has no connection with floating point arithmetic. It is dismissed in a few lines of Tang's paper, since if one doesn't insist on a fully formal proof, then maximizing a smooth function over a closed interval is a straightforward matter for a numerical programmer. However, justifying such a result by a formal proof is much harder. We took several weeks of work, becoming diverted into the necessary proofs of various results about polynomial elimination. This is described in more detail in a separate paper (Harrison 1997b), but in summary the approach is as follows.

We want to arrive at reasonably sharp bounds for $e^x - (1 + p(x))$ over a suitable range. First, we find a truncated Taylor series to approximate e^x to an accuracy well beyond that we are interested in. Thus, the problem is reduced to bounding a *polynomial*, which is a tractable problem. We just need to locate the points of zero derivative. The main difficulty is that one must prove that *all* such points have been located; in general it may be fewer than the degree of the polynomial. Thus we need to prove formally how many (real) roots a polynomial has, and to do this we formalize Sturm's theorem on polynomial remainder sequences (Benedetti and Risler 1990).

It is worth noting that the only real error we have found in Tang's proof occurs here. He bounds the polynomial approximation accuracy over the interval $[-0.010831, 0.010831]$, with the implicit assumption that this is the limit of R, or to be precise, of $Val(R_1) + Val(R_2)$. However, in the case of single precision arithmetic this is not quite correct. For example if X has the hex representation $423708C0$ (real value about 45.76, and `float(0,132,3606720)` in the HOL formalization) the corresponding R already exceeds this slightly, while for $435C0524$ (value about 220.02, and HOL `float(0,134,6030628)`) the magnitude of R has risen to over 0.010833. (The latter example is only significant if one performs bias adjustment, since its exponential is out of range, but the first is well within range, and there are several other such counterexamples.) A naive error analysis gives a bound of 0.010844, which is the range we use; this could be improved by a more delicate analysis of how multiples of Inv_L round, but this hardly affects the overall error. We arrive at an error bound of $\frac{24}{27}2^{-33}$ rather than $\frac{23}{27}2^{-33}$ which could be proved over the narrower interval assumed by Tang. By the way, it may be the case that Tang's polynomial coefficients could be improved given the new interval.

6.4 Rounding errors

Though this occupies the largest part of Tang's error analysis, it is all a routine application of earlier lemmas. This is a little laborious since it has to be repeated for about a dozen arithmetic operations, but it is far from difficult. We organize things slightly differently from Tang, and exploit HOL's programmability to compose errors appropriately. For example, the calculation of P involves the composition of 41 error terms, many of which are the product of several others, and it would be quite tedious to do this by hand. Our analysis descends all the way to the rounding error in calculating Q, while Tang simply says that because of its small size, 'the rounding errors accumulated in its calculation are practically zero'. While this is quite true, justifying this intuition formally is harder than simply bounding that error in the usual way.[2] Anyway, one might argue that 'error bounds' should be just that — guaranteed upper bounds.

We arrive at a bound for the rounding error in P of $\frac{11}{20}2^{-30}$. Though this is larger than the error Tang assumed, the overall error bounds in $E1$ that we get are actually tighter. 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

[2] Arguably this is a general problem in trying to achieve a high level of formalization in applications of scientific theories — it can be harder to justify the simplifying assumptions rigorously than to avoid them. For example, think of some of the approximations commonly used in mechanics.

are $0.5125ulp$ and $0.5338ulp$ respectively (see the annotation given earlier for the HOL statement). The first is lower because we manually observed that in this case, we must have $Ival(J) = 0$ and then most of the arithmetic operations involved in calculating P in terms of Q are exact. The better bound for the second, however, results purely from HOL's mechanical application of the theorems about error bounds. It seems paradoxical that we get a better final result despite taking into account more errors, but the explanation seems to be that we avoid inserting any great 'safety margin' in results to compensate for neglected errors.

In some ways, our argument is also more natural than Tang's in that when deciding on the binary intervals in which results lie, we carefully separate abstract mathematical values from their floating point approximations and deal always with the ones that are actually relevant. For example, we case split over the binary intervals for the computed output $E1$, since this determines the value of an ulp in the result. We deduce that if $Val(E1)$ lies in $[\frac{1}{2}, 1)$ then $j = 0$ and $Val(S_{Lead}(J)) + Val(S_{Trail}(J) + (S_{Lead}(J) + S_{Trail}(J))P) < 1$, where $j = Ival(J)$. This is then used in the remaining error analysis. By contrast, Tang considers whether $j = 0$ first and then whether $r < 0$, where r is the exact variant of R, that is, $Val(X) - Ival(N)\frac{ln(2)}{32}$. This necessitates a nontrivial additional argument to see that, for example, r and $Val(R1) + Val(R2)$ always lie in the same binary interval.

The last line of the algorithm simply scales $E1$ by 2^M to yield the final result. This is close to being simple. If the result is normalized, then the scaling is exact and the maximum error is still $0.5338ulp$. If, on the other hand, the result is denormalized, we can get an extra error of $0.5ulp$ due to rounding, but the corresponding ulp is at least twice as large as it would be if the result could be scaled exactly. Thus the maximum error is $0.5 + \frac{0.5338}{2}ulp$. However there are subtleties lurking where overflow and underflow are concerned.

6.5 Overflow

Our specification makes quite a strong statement about the overflow behaviour of the algorithm.[3] We assert that overflow occurs (i.e. the result is $+\infty$) if and only if the true exponential exceeds the standard overflow threshold. (Tang doesn't make any similarly precise claim.) It is clear that overflow occurs precisely when our approximate exponential exceeds the overflow threshold, and we are left to prove that these are sufficiently close that either one overflows if the other does.

The proof proceeds by contradiction. 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:

[3] We do not follow Tang in performing 'bias adjustment' in the case of overflow. This would be very easy to add, given that everything is tightly bounded until the final scaling by 2^M, but since this bias adjustment procedure is not specified in the IEEE standard, we don't bother.

```
|- 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 by carefully approximating $ln(threshold)$, easily done using the known approximation to $ln(2)$, we discover that this is impossible: there is no such X. The reason is that $ln(threshold)$ is straddled by two floating point values, `float(0,133,3240471)` and `float(0,133,3240472)`, and is more than 2^{-22} from either of them, although not that much further from the latter. Quite naively, we could count ourselves unlucky if this property failed, since

$$e^{X(1+\delta)} = e^X e^{X\delta} \approx e^X(1 + X\delta)$$

In other words, the relative change in the output is about X times the relative change in the input. Since X at this point is around 2^6, we might expect a probability of about 2^{-5} that the true exponential lies dangerously close (within just over $0.5ulp$) to the overflow threshold.

6.6 Underflow

There is a similar dangerous case to be ruled out at the other end of the scale. We have said that the maximum error from scaling is $0.5ulp$, but in compensation an ulp is at least twice as large relative to the result as it would otherwise be. However our specification, following Tang, asserts that this situation only arises when the final result E is denormalized. We need to rule out the possibility that $E1$ loses a bit in scaling yet then rounds back up to a normalized result. This can only happen when the exact value of $2^M E1$ is $2^{-126}(1-2^{-24})$. Now $E1$ could not, in this case, lie in the interval $[1, 2)$, since we have proved earlier that it cannot exceed about $2 - 2^{-6}$ (see the annotation shown earlier). We must therefore have $E1 \in [\frac{1}{2}, 1)$ and so $M = -126$. The problem is reduced to showing that no floating point value X has an exponential within 2^{-150} of $2^{-126}(1 - 2^{-24})$.

This is susceptible to reasoning very similar to that in the case of overflow. Abbreviating $t = 2^{-126}(1 - 2^{-24})$, we want to show that $ln(t)$ is too far away from any representable value. In fact the situation is even better than before: $ln(t)$ is straddled by `float(1,133,3058767)` and `float(1,133,3058768)` and at least 2^{-19} from either of them; once again, a distance of 2^{-22} would suffice. Hence we get the final result.

7 Conclusions and methodological remarks

When describing verification efforts, there is a tension between impressing with our persistence, and demonstrating the maturity and usability of the tools chosen. While we are not averse to making such points, the main message we want to communicate is that verifications of this nature are, with a little effort, comfortably within the state of the art, and as such are well placed for industrial exploitation.

We believe that verifications of this sort are well worthwhile. Although we have found Tang's error analysis to be correct in essentials, we have found one small slip and have located a few subtle corners in the proof that a less careful worker than Tang might easily have overlooked. One of the commonest errors in postulated theorems is the forgetting of special or degenerate cases. (IEEE floating point is particularly rich in such cases, e.g. NaNs, negative zeros etc.) Using a mechanical theorem prover means we are never allowed to miss these things. On top of that, we derive the mathematics with the greatest rigour and integrate it with the verification, so it can be relied on that we have not misapplied some 'book' result such as the error in truncating a Taylor series.

Apart from standard real analysis, some parts of this correctness proof (Harrison 1997b) needed a lot of additional pure mathematics, e.g. Sturm's theorem. So it is a mistake to believe that to get concrete results, the only theorems about the reals needed are a few algebraic and order-theoretic banalities: sometimes one needs much more. Even so, these results are often quite concrete and combinatorial, in contrast to the clean, abstract results on often sees in mathematics texts, and are typically much harder to prove formally. To give an ad hoc comparison, the work here took about 100 times as much work as a HOL proof by us of M. H. Stone's theorem that a metrizable topological space is paracompact, a result arguably of greater mathematical depth.[4] However, while the result described here is the culmination of about 3 months of hard work, most of this was devoted to getting the IEEE-754 formalization right and proving general results on polynomial approximation and floating point error analysis. This is all re-usable, so we believe other verifications of this general type could now be cranked out reasonably quickly, say in one or two weeks each.

In this algorithm, we have taken IEEE-compliant addition, multiplication etc. as given. However, it would be perfectly possible to specify instead the kind of special extra precise arithmetic that might be implemented in hardware. For example, a simple CORDIC example we dealt with some time ago is based on this approach (Harrison 1996b). Indeed, we could attempt to verify some implementations of the IEEE primitives in terms of their low-level components. The proofs would presumably be quite simple compared with those tackled here.

Although the present algorithm could be verified by exhaustive testing, our proof scales straightforwardly to double, extended or perhaps even generic versions of the operation, whereas exhaustive testing seems not to. In particular, Tang sketches a proof for the double precision case, and it would be a straightforward mechanical process to adapt the present proof. (We might do this, but on reflection will probably make some improvements, as noted later.) This is an example of a general phenomenon: verification has more modularity than testing, i.e. it is often possible to re-use and modify proofs, or to link together proofs for submodules of a system, whereas this can be problematic for some other ap-

[4] This theorem was suggested on the QED mailing list on 17th June 1994 by Andrzej Trybulec as an interesting case study in the relative power and flexibility of theorem proving systems — see `ftp://ftp.mcs.anl.gov/pub/qed/archive/56` and `ftp://ftp.mcs.anl.gov/pub/qed/archive/66`.

proaches. However, testing does have the advantage, where it is applicable, of giving (in principle) tighter error bounds. That is, the various errors that are accumulated may be correlated in ways too subtle to analyze mathematically. However we don't believe that would make a substantial difference to the error bounds here.

We have assumed a correct algorithm. But is testing or verification better for detecting errors? Testing has the advantage of indicating counterexamples directly. By contrast, in verification, one simply finds that a proof does not go through as hoped, as in our attempts to prove Tang's bound on R. This doesn't (in this case or typically) constitute a *disproof*, but only motivates a more directed search for counterexamples. We found our counterexamples by testing numbers where the rounding to an integer leaves the greatest possible error of $0.5ulp$. On the other hand, while verification doesn't pinpoint *counterexamples*, it does pinpoint *mathematical errors* more directly. To go from a failure in testing to an analysis of the problem, one needs to trace through the counterexample to see where things go wrong. So we might say that testing is better for finding *when* things go wrong, and verification for finding *why and where* things (might) go wrong.

As for other theorem provers, there is hardly an alternative to some version of HOL for verifications of this kind at present. One needs available a large library of results from real analysis — for example we have used the mean value theorem, Taylor's theorem and common properties of the exponential function. One also needs a system that is reasonably 'heavyweight'; biased towards big, ugly verification proofs rather than the elegant solution of high-level mathematical problems. While some systems have one (e.g. Mizar) and some have the other (e.g. NQTHM), no other system that we are aware of combines these strengths. Probably the best alternative would be PVS, which has already benefited from some excellent work in formalized mathematics (Dutertre 1996) and floating point verification (Miner 1995; Miner and Leathrum 1996). Its type system offers some extra features over HOL's that might be helpful in giving a clear specification; for example, it is not necessary to distinguish between $\mathbb{N}$ and $\mathbb{R}$.

In future verifications, we can make better use of HOL's programmability. As it stands, we automated certain routine steps, e.g. the application of the valuation function Val to particular floating point values, the accumulation of rounding errors in expressions, and the evaluation of Taylor approximations. However, much more could be done and from the perspective gained as a result of this proof, we can identify the most important things to automate. We have used HOL's existing tools to prove innumerable trivial but tedious results of linear arithmetic automatically, and this was almost indispensable. But one of the most tiresome aspects of the proofs was that similarly simple results with a nonlinear component need to be proved manually. Similar experiences are reported by Miner and Leathrum (1996) when using PVS. However we have now implemented experimental tools to automate these steps, including the semi-automatic determination of the signs of product terms, many of which are of the form 2^k and are hence trivially strictly positive. We believe these tools would have been invaluable in this verification, and should streamline future efforts. We also intend to clean up, fill out and generalize the most useful-looking lemmas; in the present verification almost all of these were accreted ‘by need’ without any systematic plan.

Our insistence on reducing everything to logical first principles gives us the highest confidence in the final result, and has not been a disaster from the point of view of efficiency. There is one unfortunate exception: arithmetic. Performing arithmetic, particularly multiplication, of large numbers by proof is very time-consuming. So much so that the entire proof takes about 12 hours to run; this is cut to about 2 hours if addition and multiplication are performed as primitives using CAML's native bignums, rather than via logical decomposition. It might be a pragmatic *necessity* to do this for double precision or extended precision verifications; these are structurally identical and no harder for the user, but involve calculation with still larger numbers. The main problem areas are the evaluation of Sturm sequences and the accurate approximation of $ln(2)$ and $2^{\frac{j}{32}}$. On the other hand, most of the arithmetic operations could be done in a more sophisticated way using approximations tailored to the required accuracy rather than relying on exact rational arithmetic (Harrison 1996b).

There is an argument though, that forcing the user to think carefully about something as basic as arithmetic in order to make it acceptably efficient is not conducive to great productivity. Perhaps it should simply be made primitive. The ACL2 prover makes a virtue of its ability to mix efficient calculation in with the proof process. Our verification gives a good illustration of how important this can be. Of course ACL2 is hardly suited to verifications of this type since it has no notion of real number, and would therefore be unable to formalize some of the higher level details directly. However ACL2 has been used for interesting work in verifying a more elementary floating point operation, namely division (Moore, Lynch, and Kaufmann 1996; Brock, Kaufmann, and Moore 1996). This work in ACL2 was actually carried out in cooperation with the designer of the algorithm at AMD.

Acknowledgements

Thanks are due to Mark Aagaard, who suggested this algorithm for verification, and to Clemens Ballarin whose comments on a draft of this report have led to several improvements.

References

Barratt, M. (1989) Formal methods applied to a floating-point system. *IEEE Transactions on Software Engineering*, **15**, 611–621.

Benedetti, R. and Risler, J.-J. (1990) *Real algebraic and semi-algebraic sets*. Hermann, Paris.

Brock, B., Kaufmann, M., and Moore, J. S. (1996) ACL2 theorems about commercial microprocessors. See Srivas and Camilleri (1996), pp. 275–293.

Dutertre, B. (1996) Elements of mathematical analysis in PVS. See von Wright, Grundy, and Harrison (1996), pp. 141–156.

Goldberg, D. (1991) What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, **23**, 5–48.

Gordon, M. J. C. (1989) Mechanizing programming logics in higher order logic. In Birtwistle, G. and Subrahmanyam, P. A. (eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 387–439. Springer-Verlag.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Harrison, J. (1996a) HOL light: A tutorial introduction. See Srivas and Camilleri (1996), pp. 265–269.

Harrison, J. (1996b) Theorem proving with the real numbers. Technical Report 408, University of Cambridge Computer Laboratory.

Harrison, J. (1997a) Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory.

Harrison, J. (1997b) Verifying the accuracy of polynomial approximations in HOL. In Gunter, E. L. and Felty, A. (eds.), *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, Volume 1275 of *Lecture Notes in Computer Science*, Murray Hill, NJ, pp. 137–152. Springer-Verlag.

Miner, P. S. (1995) Defining the IEEE-854 floating-point standard in PVS. Technical memorandum 110167, NASA Langley Research Center, Hampton, VA 23681-0001, USA.

Miner, P. S. and Leathrum, J. F. (1996) Verification of IEEE compliant subtractive division algorithms. See Srivas and Camilleri (1996), pp. 64–78.

Moore, J. S., Lynch, T., and Kaufmann, M. (1996) A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division algorithm. Unpublished; available on the Web as `http://devil.ece.utexas.edu:80/~lynch/divide/divide.html`.

Pratt, V. R. (1995) Anatomy of the Pentium bug. In Mosses, P. D., Nielsen, M., and Schwartzbach, M. I. (eds.), *Proceedings of the 5th International Joint Conference on the theory and practice of software development (TAPSOFT'95)*, Volume 915 of *Lecture Notes in Computer Science*, Aarhus, Denmark, pp. 97–107. Springer-Verlag.

Srivas, M. and Camilleri, A. (eds.) (1996) *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, Volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag.

Tang, P. T. P. (1989) Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, **15**, 144–157.

von Wright, J., Grundy, J., and Harrison, J. (eds.) (1996) *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style