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
jrh@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/jrh

8th September 1997

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.

The development described here includes, apart from the proof itself, a formalization of IEEE arithmetic, a mathematical semantics for the programming language in which the algorithm is expressed, and the body of pure mathematics needed. All this is developed logically from first principles using the HOL Light prover, which guarantees strict adherence to simple rules of inference while allowing the user to perform proofs using higher-level derived rules.

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). Indeed, it is sometimes difficult to say what it *means* for a floating point operation to be 'correct'. We hope here to provide some answers to this question, and show how we can guarantee the correctness of a realistic floating point algorithm according to one of these criteria.

Traditional techniques for ensuring correctness rely on extensive testing. Actually, in the case of unary functions over single precision floating point numbers (of

[*]Supported by the EPSRC research grant 'floating point verification'.

which there are 'only' 2^{32} — actually slightly fewer) it is possible to run completely exhaustive tests, though the difficulty of efficiently generating correct results to act as the canon should not be minimized.[1] For binary operations or for double or extended precision numbers, this kind of testing is hardly feasible. One can still run very large test suites that try to exercise the cases felt to be error-prone, but this can no longer guarantee correctness.

In order to do better, we advocate formal verification using a mechanical theorem prover. Such provers can encompass essentially all of classical mathematics, including real numbers and real analysis, giving two great advantages:

- We can specify the correctness of floating point operations in terms of abstract real numbers. After all, the whole *raison d'être* of floating point numbers is that they are approximations to reals.
- The correctness of many floating point algorithms depends on some quite sophisticated mathematics. Using a theorem prover, one can avoid errors either in the pure mathematics or in its incorrect application to the problem in hand.

The idea of mathematical proofs of computer system correctness has in the recent past attracted a certain amount of controversy — see Barwise (1989) for a good discussion. So perhaps we should not neglect to make the ritualistic disclaimer that our verification is of an *algorithm*, expressed in a precise mathematical formalism, not of any actual implementation. Nevertheless, as will be seen below, we have taken some pains to express the algorithm in something close to typical programming and hardware description languages, in the hope of minimizing this semantic gap. Similarly, the results must be understood in the context of the definitions on which they rest, and these may fail to capture one's intentions. We hope that we have not used any faulty or even controversial definitions in what follows.

As for the relative value of a formalized proof of this kind compared with an informal proof, we believe, *pace* DeMillo, Lipton, and Perlis (1979), that a HOL proof such as ours should have greater persuasive power. This is not to say that our HOL proof script is the most convenient way of communicating the essence of the proof to people — if we believed that we would emit our ASCII text verbatim rather than write this article. But when the processes underlying the HOL proof script are understood, we hope it will *compel belief*, and that is what we aim at first and foremost. This property is quite orthogonal to the question of a proof's 'surveyability' and its susceptibility to the usual social processes, just as the question of whether a computer can think is not relevant to whether a particular program can beat one at chess. Having said that, we do consider the surveyability of formal proof outlines important,[2] and have recently (Harrison 1996a) been experimenting with more readable and declarative proof outlines inspired by the Mizar system (Trybulec 1978).

While large ill-defined software systems are far from being verifiable, at least at present, there is no doubt that present verification techniques are applicable to some significant problems. In particular, basic floating point operations seem to be a good target. They embody relatively small but mathematically sophisticated algorithms. By focusing on them with a high level of formality, we can provide a secure foundation for higher level work in numerical analysis, even when that is of itself out of reach of a formal treatment.

[1] The original Occam floating point code used by Inmos was tested against a standard numeric coprocessor. One of the observed discrepancies turned out to be a bug not in the code but in the coprocessor.

[2] Pollack (1997) points out that even though a formal proof may not itself be 'surveyable', a computer program capable of surveying it might be. In a sense, computers can furnish a form of leverage.

2 HOL Light

HOL Light (Harrison 1996b) is our own version of the HOL prover (Gordon and Melham 1993), which is itself descended from Edinburgh LCF (Gordon, Milner, and Wadsworth 1979) via Cambridge LCF (Paulson 1987). HOL Light maintains most of the general principles underlying its ancestors, but attempts to be more logically coherent, elegant and usable. It is written entirely in CAML Light (Weis and Leroy 1993), giving it advantages of portability and low resource usage compared with its ancestors, which are based on LISP or Standard ML.

HOL Light is simply a large CAML program that defines data structures to represent logical entities, together with a suite of functions to manipulate them in a way guaranteeing soundness. The most important data structures belong to one of the datatypes `hol_type`, `term` and `thm`, which represent types, terms (including formulas) and theorems respectively. The user can write arbitrary programs to manipulate these objects, and it is by creating new objects of type `thm` that one proves theorems. HOL's notion of an 'inference rule' is simply a function with return type `thm`.

In order to guarantee logical soundness, however, all these types are encapsulated as abstract types. In particular, the only way of creating objects of type `thm` is to apply one of a dozen or so very simple primitive inference rules. Thus, whatever the circuitous route by which one arrives at it, the validity of any object of type `thm` rests only on the correctness of the rather simple primitive rules (and of course the correctness of CAML's type checking etc.). For example, one of HOL's primitives is the rule of Modus Ponens,

$$\frac{\Gamma \vdash p \Longrightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{MP}$$

This allows one to make the following logical step: if under assumptions Γ one can deduce $p \Longrightarrow q$ (that is, p implies q), and under assumptions Δ one can deduce p, then one can deduce that from all the assumptions together, $\Gamma \cup \Delta$, that q holds. If the two starting theorems are bound to the names `th1` and `th2`, then one can apply the above logical step in HOL and bind the result to `th3` via:

```
let th3 = MP th1 th2;;
```

One doesn't normally use such low-level rules much, but instead interacts with HOL via a series of higher-level derived rules, using built-in parsers and printers to write terms in a more natural syntax. For example, if one wants to bind the name `th6` to the theorem of real arithmetic that when $|c - a| < e$ and $|b| \leq d$ then $|(a + b) - c| < d + e$, one simply does:

```
let th6 = REAL_ARITH
  `abs(c - a) < e ∧ abs(b) <= d
   ⟹ abs((a + b) - c) < d + e`;;
```

If the purported fact in quotations turns out not to be true, then the rule will fail by raising an exception. Similarly, any bug in the derived rule (which represents several dozen pages of code written by the present author) would lead to an exception.[3] But we can be rather confident in the truth of any theorem that is returned, since it must have been created via applications of primitive rules, even though the precise choreographing of these rules is automatic and of no concern to the user. What's more, users can write their own special-purpose proof rules in the

[3] Or possibly to a true but different theorem being returned, but this is easily guarded against by inserting sanity checks in the rules.

same style when the standard ones seem inadequate — HOL is fully programmable, yet retains its logical trustworthiness when extended by ordinary users.

Among the facilities provided by HOL is the ability to organize proofs in a mixture of forward and backward steps, which users often find more congenial. The user invokes so-called *tactics* to break down the goal into more manageable subgoals. For example, the standard HOL proof of an elementary fact of number theory that addition of natural numbers is commutative is written as follows (the symbol ∀ means 'for all'):

```
let ADD_SYM = prove
 (`∀m n. m + n = n + m`,
  INDUCT_TAC THEN
  ASM_REWRITE_TAC[ADD_CLAUSES]);;
```

The tactic INDUCT_TAC uses mathematical induction to break the original goal down into two separate goals, one for $m = 0$ and one for $m + 1$ on the assumption that the goal holds for m. Both of these are disposed of quickly simply by repeated rewriting with the current assumptions and a previous, even more elementary, theorem about the addition operator. The identifier THEN is a so-called *tactical*, i.e. a function that takes two tactics and produces another tactic, which applies the first tactic then applies the second to any resulting subgoals (there are two in this case).

For another example, we can prove that there is a unique x such that $x = f(g(x))$ if and only if there is a unique y with $y = g(f(y))$ using a single standard tactic MESON_TAC, which performs model elimination (Loveland 1968) to prove theorems about first order logic with equality. As usual, the actual proof under the surface happens by the standard primitive inference rules.

```
let WISHNU = prove
 (`(∃!x. x = f (g x)) ≡ (∃!y. y = g(f y))`,
  MESON_TAC[]);;
```

These and similar higher-level rules certainly make the construction of proofs manageable whereas it would be almost unbearable in terms of the primitive rules alone. Nevertheless, we want to dispel any false impression given by the simple examples above: nontrivial proofs, as are carried out in the work described here, often require long and complicated sequences of rules. The construction of these proofs often requires considerable persistence. Moreover, the resulting proof scripts can be quite hard to read, and in some cases hard to modify to prove a slightly different theorem. One source of these difficulties is that the proof scripts are highly procedural — they are, ultimately, CAML programs, albeit of a fairly stylized form. As mentioned above, we have recently been experimenting with declarative scripts, which promise to be more readable, and more maintainable; perhaps even more writable.

In presenting HOL theorems below, we will use standard symbols for the logical operators, as we have in the above examples, but when actually interacting with HOL, ASCII equivalents are used:

Standard symbol	ASCII version	Meaning
$\perp$	`F`	Falsity
$\top$	`T`	Truth
$\neg$	`~`	Not
$\wedge$	`/\`	And
$\vee$	`\/`	Or
$\Longrightarrow$	`==>`	Implies
$\equiv$	`=`	If and only if
$\forall$	`!`	For all
$\exists$	`?`	There exists
ε	`@`	Hilbert choice
λ	`\`	Lambda abstraction

The term $\lambda x.\ t[x]$ means 'the function that maps each x to $t[x]$', while $\varepsilon\ P$ denotes 'some x such that $P(x)$'. For more on the HOL logic, see Gordon and Melham (1993), while for more details on the development of the HOL mathematical theories of real numbers, real analysis and transcendental functions on which this verification rests, we refer the reader to our PhD thesis (Harrison 1996c). Most of the operations on 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.

3 Correctness criteria for floating point operations

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,[4] 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 intend to specify the correctness of a floating point operation 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. For example, the correctness of a floating point operation EXP is specified by comparing it with its ideal counterpart *exp*. 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':

[4] 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.

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

Analogously, for a binary function, say multiplication, we can consider the relationship between $v(a)v(b)$ and $v(\mathrm{MUL}(a,b))$. But just what relationship are we to demand between the exact and computed values (e.g. $exp(v(a))$ and $v(\mathrm{EXP}(a))$) as our criterion for correctness? This is a delicate question, and we examine it at length.

3.1 Round to nearest

It seems that the most stringent specification we can give is that no floating-point number is closer to the true mathematical answer. For example, in the case of a multiplication operation MUL this means:

$$\forall a, b \in \mathbb{F}.\ \neg \exists c \in \mathbb{F}.\ |v(c) - v(a)v(b)| < |v(\mathrm{MUL}(a,b)) - v(a)v(b)|$$

Actually, though, this is too lax to specify the result of a floating point operation completely, since it may happen that there are two floating point values equally close to the exact result. Humans working in decimal are usually taught to round $0.5000\ldots$ up to 1, but the IEEE (1985) standard for binary floating point arithmetic mandates a (default) rounding mode of 'round to even':

> An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered.

Fixing the choice between such alternatives has the merit that operations will behave identically on all implementations, even if they use quite different (correct!) algorithms. The IEEE Standard demands that the operations of addition, subtraction, multiplication, division, and square root be performed as if done with full accuracy then rounded using this scheme or whichever of the others is in force (see later). This approach based on the exact mathematical result ensures that the operations share important characteristics of the abstract counterparts. For example, multiplication is:

- Symmetric, i.e. $\mathrm{MUL}(a,b) = \mathrm{MUL}(b,a)$
- Monotone, i.e. if $0 \le v(a) \le v(a')$ and $0 \le v(b) \le v(b')$ then we also have $v(\mathrm{MUL}(a,b)) \le v(\mathrm{MUL}(a',b'))$.

Empirical studies suggest that 'round to even' avoids a tendency for floating point results to drift systematically. Using the 'human style' rounding, there is a tendency to drift upwards under successive operations, presumably because the situation of being midway between two representable values happens quite often.

For example, when adding normalized numbers with the same sign and identical exponents, exactly one bit, which is often 1, must be discarded. (By contrast, subtraction of such pairs of numbers is always exact.)

3.2 Bounded error

The above correctness criterion is sometimes unrealistically stringent for other functions such as the transcendentals *exp*, *cos*, *exp*, *ln* etc; this is probably why the IEEE standard does not discuss such functions at all. The difficulty is known as the 'table maker's dilemma', and arises because 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. For example, a value x one is approximating may be exactly the rational number 3.15. This being the case, knowing for any given $n \in N$ that $|x - 3.15| < 10^{-n}$ does not help to decide whether 3.1 or 3.2 is the correctly rounded result to one decimal place.

In the case of the common transcendental functions like *exp*, there are results of number theory assuring us that for nontrivial rational arguments the result is irrational. In fact for nonzero algebraic $x \in \mathbb{C}$ it follows from a classic theorem of Lindemann (1882) that all the following are not just irrational but transcendental:[5] $exp(x)$, $sin(x)$, $cos(x)$, $tan(x)$, $sinh(x)$, and $cosh(x)$, and for $x \neq 1$ too, $cos^{-1}(x)$ and $ln(x)$ — Baker (1975) gives a proof. And all floating-point values *are* rational of course, at least in conventional representations, though not in novel schemes like the exponential towers proposed by Clenshaw and Olver (1984). However the appropriate bounds on the evaluation accuracy required to ensure correct rounding in all cases may be impractically hard to find analytically. Exhaustive search is hardly an option for double precision, though perhaps more systematic ways are feasible. Even if the evaluation accuracy bounds are found, they may be very much larger than the required accuracy in the result. Goldberg (1991) gives the illustrative example of:

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

To round the result correctly to 4 decimal places, one needs to approximate it to about 10^{-9}. Insisting on exact rounding therefore tends to imply a commitment to substantially longer-than-usual computations in special cases. This is possible, but it is costly in hardware (so is most likely to be implemented via software traps) and means that execution times and space usage can no longer be guaranteed, or at least so sharply bounded.

In summary then, it is usually appropriate to settle for a weaker criterion. Notably, we can simply insist that the error is within some bound. One is seldom interested in absolute error, since floating point numbers vary widely in magnitude, so in practice one measures the error in a value relative to the value itself. Either one can consider the relative error, i.e. the ratio of the error to the value, or the error in terms 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 $\leq 0.5\ ulp$.

[5] A number is said to be algebraic if it is the root of a polynomial with integer coefficients, e.g. $\sqrt{2}$ is a root of $x^2 - 2$, and to be transcendental otherwise. Rational numbers are roots of polynomials of degree 1.

3.3 Error commensurate with likely input error

Even guaranteeing small relative errors, while quite possible, can be rather difficult. For example, consider the evaluation of $sin(x)$ where x is just below the largest representable number in IEEE double precision, about $2^{1024} \approx 1.8 \times 10^{308}$. Most underlying algorithms for sin, e.g. Taylor series, only work, or only converge reasonably quickly, for fairly small arguments. Therefore the usual first step in calculating $sin(x)$ is to perform an appropriate range reduction, e.g. finding an x' with $-\pi \leq x' < \pi$ and $x' - x = 2n\pi$ for $n \in \mathbb{Z}$. However in this case, performing the range reduction accurately is not straightforward. Simply evaluating $x/(2\pi)$, rounding this to an integer n and then evaluating $x' = x - 2n\pi$, where the operations are performed with anything like normal accuracy, is going to return 0, because n is far too large a number for its fractional part to make an appearance in the bit sequence, making the rounding operation trivial. So if accurate rounding is required, it's necessary to store $\frac{1}{\pi}$ or some such number to over a thousand bits, and perform the range reduction calculation to that accuracy (Payne and Hanek 1983).

Since this sort of super-accurate computation need only be kicked in when the argument is large, a situation that one hopes will, in the hands of a good numerical programmer, be exceptional, it need not affect average performance. However in a hardware implementation in particular, there may be a significant cost in chip area which might be put to better use. It seems that many designers do not make the required effort. Ng (1992) shows the huge disparities between (then) current computing systems on $sin(10^{22})$: only a very few systems, notably the VAX and SPARC floating point libraries and certain HP calculators, give the right answer.

One can certainly defend the policy of giving inaccurate answers in such situations, and this leads us to a new notion of correctness. Floating point calculations are generally not performed in number-theoretic applications. The *inputs* to floating point calculations are usually inexact, because they are necessarily approximate measures of physical quantities, because they are exact quantities that don't have an exact floating point representation (e.g. $\frac{1}{10}$ in IEEE binary arithmetic), or because they are the rounded results of previous floating point calculations. One might expect that the average floating point value is inaccurate, compared with some abstract ideal, by at least $0.5ulp$. Accordingly, we may say that a function is correct if the disparity between its output and the true mathematical answer could be explained by a very small (say $0.5ulp$) perturbation of the input:

$$\exists \eta.\ |\eta| \leq 0.5\ ulp_x \wedge f(v(x) + \eta) = v(F(x))$$

There is an obvious analogy between this approach and backward error analysis (Wilkinson 1963). Nevertheless, as pointed out by Kahan,[6] one sometimes wants the appropriate 'input perturbations' for various functions to be correlated in some way. For example, when calculating:

$$f(x) = \begin{cases} \frac{1}{2} & \text{if } x = 1 \\ -atn(ln(x))/acos(x)^2 & \text{if } x < 1 \\ atn(ln(x))/acos(x)^2 & \text{if } x > 1 \end{cases}$$

replacing $ln(x)$ and $acos(x)$ by $ln(x')$ and $acos(x'')$ for close but uncorrelated x' and x'' can lead to grave inaccuracy as $x \to 1$, even though the function as a whole is well-behaved there.

[6] See, e.g. the notes for CS 267, available on the Web as `http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/cs267fp.ps`.

3.4 Probabilistic and mixed criteria

One can always modify any of the above criteria, most naturally the one demanding perfect rounding, in a probabilistic way. That is, one may say that a function is correct if the correctness criterion is met in a sufficient proportion, e.g. 99.9%, of cases. Strictly speaking, this isn't a probabilistic measure because the applications of the operations are unlikely to be evenly distributed, but the probabilistic interpretation is natural. This can easily be formalized using cardinalities of sets, e.g:

$$card\ \{a \in \mathbb{F} \mid |exp(v(a)) - v(\mathrm{EXP}(a))| > 0.5ulp(\mathrm{EXP}(a))\} < card(\mathbb{F})/1000$$

This might be considered a bit unsatisfactory, since it sanctions wild inaccuracy in exceptional cases. One can however combine this with a definite but worse than $0.5ulp$ bound on the accuracy in all cases. See Gal (1986) for an example of this kind of correctness result for a floating point logarithm function not dissimilar in structure to the algorithm we are concerned with here.

4 Our implementation language

In early versions of this work (Harrison 1995), floating point algorithms were expressed directly by primitive recursive functions in the HOL logic. However such a formalization has only an indirect relationship with any actual implementation of the algorithm, either in hardware or software. Now we opt to express the algorithms in a simple imperative 'while language'. Using a recognizable programming language in this way offers two benefits.

First, the language corresponds quite closely to the typical pseudocodes often used to describe algorithms. In fact, there is a view, expressed for example by Dijkstra (1976), that a programming language should be thought of first and foremost as an algorithm-oriented system of mathematical notation, and only secondarily as something to be run on a machine. Our HOL embedding fits nicely with this principle since programming constructs are given a simple semantics using objects in the HOL logic.

Secondly, our little language can naturally be seen as a subset either of programming languages like C or Ada, or of high-level hardware description languages such as Verilog, VHDL and Hardware C; hence it corresponds quite directly to implementations either in software or hardware. There are projects at the University of Cambridge Computer Laboratory looking at formal HOL-based semantics for the C programming language (Norrish 1997) and the Verilog hardware description language (Gordon 1995), as well as a new project concerned with Handel (Page 1996). We hope in due course to make a more formal connection with at least some of these efforts.

The language is given a simple operational semantics, and from that, weakest preconditions, total correctness rules and verification condition generators are derived. This is not novel in any significant detail, being based on work of Gordon (1989), von Wright, Hekanaho, Luostarinen, and Långbacka (1993) and Nipkow (1996).[7] We expect that the few constructs we use in this work are quite familiar without any formal treatment. Note that expressions map down to terms in the HOL logic, and so in the concrete syntax, parentheses may be used to enforce grouping, just as one uses braces in C and `begin` ... `end` in Pascal. Program fragments

[7] One unusual feature is that verification conditions are generated *forward* from the precondition P for assignment statements that, viewed as state mappings, do not affect the precondition ($p \circ f = p$) and are idempotent ($f \circ f = f$). For programs like the present one with many variables assigned just once, this tends to make the verification conditions more intuitive.

can be parsed and prettyprinted in a readable form, enclosed within '`var` *variables* ... `end`'. The `var` construct declares those variables that are to be treated as program variables. Any other identifiers are regarded as logical 'variables' in the usual sense.

The special command '$\{expression\}$' is called an *assertion*. It checks a condition, and if it is true, does nothing, while if it is false, it goes into an infinite loop. The user can specify properties that should hold at a point in the code using assertions, and these are used by HOL to generate *verification conditions* (see below). As well as these assertions, the user can annotate while-loops and do-loops with an 'invariant', which should be preserved round each iteration of the loop, and a natural number 'measure' that decreases round each iteration; the latter ensures that the loop terminates. The example below illustrates how these annotations are inserted in the concrete syntax.

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.[8] 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. For example, the following is a correctness assertion for a trivial program to calculate factorials:[9]

```
correct
T
  var x,y,n;
    x := 0;
    y := 1;
    while x < n do [invariant x <= n ∧ (y = FACT x); measure n - x]
     (x := x + 1;
      y := y * x)
  end
y = FACT n
```

The assertion is that whatever the initial state ('`T`' is always true), the program will terminate in a state with $y = n!$. Setting this as the current goal and applying the HOL verification condition generation tactic `VC_TAC` gives rise to three VCs:

```
?- x <= n ∧ (y = FACT x) ∧ x < n ∧ (X = n - x)
     ⟹ x + 1 <= n ∧ (y * (x + 1) = FACT (x + 1)) ∧ n - (x + 1) < X

?- ¬(x < n) ∧ x <= n ∧ (y = FACT x) ⟹ (y = FACT n)

?- (x = 0) ∧ (y = 1) ⟹ x <= n ∧ (y = FACT x)
```

These are all easy to prove (one or two lines of HOL tactic script), and when that is done, HOL automatically proves the initial correctness goal.

[8] HOL can automatically create verification conditions given only loop annotations (thanks to Tobias Nipkow via Mike Gordon for pointing this out to me), but sometimes the VCs are more perspicuous if the user provides guidance using assertions.

[9] This program uses abstract mathematical natural numbers, but our later examples use actual floating point numbers and finite integers.

While we consider verification condition generation to be a reasonable way to perform code verification, the underlying HOL theory does not force one to work in this way. For example, those who prefer to use stepwise refinement to develop programs from specifications (Back 1980) are free to do so.

5 Formalizing IEEE arithmetic

Increasingly, manufacturers are trying to make their floating point hardware (and software, where relevant) correspond to the IEEE (1985) standard 754 for binary floating point arithmetic. (This is to be distinguished from the IEEE standard 854, which is radix-independent, albeit in a weak sense, allowing radix 2 or 10.) We will therefore assume that our ultimate goal is to deal with IEEE 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.

5.1 Format parameters

We parametrize floating point formats by a pair of two natural numbers: the width in bits of the exponent field ($expwidth$), and the width in bits of the significand field ($fracwidth$). From these basic parameters we derive four other characteristic numbers:

- The total word length $wordlength = expwidth + fracwidth + 1$ (one bit is used for the sign of the fraction).
- The maximum exponent value $emax = 2^{expwidth} - 1$.[10]
- The bias in the exponent $bias = 2^{expwidth-1} - 1$. The exponent is stored as an unsigned number e but then treated as $e - bias$.

In HOL, we define functions to extract the primitive and derived parameters from an arbitrary format X:

```
|- expwidth (ew,fw) = ew

|- fracwidth (ew,fw) = fw

|- wordlength(X) = expwidth(X) + fracwidth(X) + 1

|- emax(X) = 2 EXP expwidth(X) - 1

|- bias(X) = 2 EXP (expwidth(X) - 1) - 1
```

[10] Note that what the standard calls E_{max} is $emax - bias - 1$.

This approach, which coincides with Barratt's, certainly covers the single and double precision formats. Strictly speaking, the exponent range and bias may be defined in some other way for the two extended formats. We could use more independent parameters to specify formats. But anyway we are only concerned in what follows with single and double precision.

5.2 Floating point numbers

The actual floating point numbers in a given format are represented by triples of natural numbers, interpreted as a sign, an exponent and a fraction. We define predicates partitioning the numbers into NaNs (NaN = not a number), infinities, normalized numbers, denormalized but nonzero numbers, and zeros. We are following the terminology of the standard in excluding zeros from the class of denormalized numbers.

```
|- is_nan(X) (s,e,f) = (e = emax(X)) ∧ ¬(f = 0)

|- is_infinity(X) (s,e,f) = (e = emax(X)) ∧ (f = 0)

|- is_normal(X) (s,e,f) = 0 < e ∧ e < emax(X)

|- is_denormal(X) (s,e,f) = (e = 0) ∧ ¬(f = 0)

|- is_zero(X) (s,e,f) = (e = 0) ∧ (f = 0)
```

Two useful derived predicates test whether a triple is valid for a given format, and if so, whether it is a finite number, i.e. not an infinity or a NaN.

```
|- is_valid(X) (s,e,f) =
      s < 2 ∧ e < 2 EXP expwidth(X) ∧ f < 2 EXP fracwidth(X)

|- is_finite(X) a =
      is_valid(X) a ∧
      (is_normal(X) a ∨ is_denormal(X) a ∨ is_zero(X) a)
```

It's also convenient to have extractors for the three fields of a floating point number:

```
|- sign (s,e,f) = s

|- exponent (s,e,f) = e

|- fraction (s,e,f) = f
```

We also define constants for a few convenient values, including the largest representable positive number in a format (`topfloat`), and the most negative number (`bottomfloat`).

```
|- plus_infinity(X) = (0,emax(X),0)

|- minus_infinity(X) = (1,emax(X),0)

|- plus_zero(X) = (0,0,0)

|- minus_zero(X) = (1,0,0)

|- topfloat(X) = (0,emax(X) - 1,2 EXP fracwidth(X) - 1)

|- bottomfloat(X) = (1,emax(X) - 1,2 EXP fracwidth(X) - 1)
```

5.3 Representation and valuation

Now we define the concrete representation for numbers, regarded as binary numerals — Miner (1995) actually makes the bitstrings explicit. Note that once again this encoding is not obligatory for the two extended formats.

```
|- encoding(X) (s,e,f) =
      s * 2 EXP (wordlength(X) - 1) + e * 2 EXP fracwidth(X) + f
```

That is, the fields are laid out with the sign as the most significant bit, the (biased positive) exponent in the middle and the fraction at the bottom. Now we specify the real number valuation of nonexceptional numbers. This is meaningless when applied to infinities and NaNs.

```
|- valof X (s,e,f) =
      if e = 0
      then --(&1) pow s * &2 / &2 pow bias(X) *
           &f / &2 pow fracwidth(X)
      else --(&1) pow s * &2 pow e / &2 pow bias(X) *
           (&1 + &f / &2 pow fracwidth(X))
```

Note that denormalized numbers and normalized numbers are treated separately. By virtue of this definition, there are no redundant multiple encodings of real numbers, except for ± 0. (Once again, redundant encodings are allowed in the extended formats according to IEEE-754, subject to a few conditions.) We define a few significant real number values. The first is intended as the real value of the largest representable number; the second is the overflow threshold for round-to-nearest, and the third is the standard notion of one *ulp* (unit in the last place) for a given floating point number:

```
|- largest(X) = (&2 pow (emax(X) - 1) / &2 pow bias(X)) *
                (&2 - inv(&2 pow fracwidth(X)))

|- threshold(X) = (&2 pow (emax(X) - 1) / &2 pow bias(X)) *
                  (&2 - inv(&2 pow SUC(fracwidth(X))))

|- ulp(X) a = valof(X) (0,exponent(a),1) - valof(X) (0,exponent(a),0)
```

5.4 Rounding

The above definition of `valof` is fundamental to what follows, as is the inverse operation of rounding, which coerces a real number into a given floating point format. This rounding is controlled by a rounding mode, specifying whether a real number is to be mapped to the nearest floating point number (using 'round to even' to choose a unique number if necessary, as described earlier), towards zero, or towards positive or negative infinity. We represent these choices in HOL via an enumerated type definition:

```
define_type
  "roundmode = To_nearest | To_zero | To_pinfinity | To_ninfinity";;
```

The actual specification of rounding is intuitively obvious, with the only subtlety being the question of whether a real number overflows and whether, if so, the rounded version should be an infinity. The standard specifies (4.1) that in the round to nearest mode, overflow occurs if the magnitude of the real number is $\geq 2^{E_{max}}(2 - 2^{-(fracwidth+1)})$, resulting in the appropriately signed infinity. The specifications for the other modes (4.2) are a bit laconic, but clear enough if one takes for granted a real line with infinities adjoined and the ordering $-\infty < finite < \infty$.[11] For example, when rounding to $+\infty$, the answer is the 'format's value (possibly $+\infty$) closest to and no less than the infinitely precise result'. This implies that if the input value strictly exceeds the largest representable finite number, it is rounded to $+\infty$, while if it is below the lowest (i.e. most negative) representable number, it should be rounded to that lowest number.

Apparently, the standard defines rounding as an operation mapping into a subset of the extended real line $\mathbb{R} \cup \{+\infty, -\infty\}$, rather than back into the actual floating point format. Our decision to map back to the actual floating point format gives us an extra choice over the sign of results that are zero after rounding. We could make a canonical choice, but instead we leave it undetermined. (When we come to the actual operations, there are case-by-case rules for the signs of zeros.) First, we define a quite general notion of rounding. The following predicate means that a is an element of the set s that provides a best approximation to x, assuming a valuation function v:

```
|- is_closest v s x a =
      a IN s ∧
      ∀b. b IN s ⟹ abs(v(a) - x) <= abs(v(b) - x)
```

However, we need something still more general to incorporate 'round to even', namely a set/predicate p that defines a set of preferred values when there are multiple 'closest' a. We define a function that picks out, using the Hilbert choice operator, a best approximation in this sense.

```
|- closest v p s x =
      εa. is_closest v s x a ∧
          ((∃b. is_closest v s x b ∧ p(b)) ⟹ p(a))
```

Now at last we define the actual rounding function for an arbitrary floating point format X. This is defined casewise on the rounding modes; in HOL terms this is a (vacuously) primitive recursive definition. We just show the clause for 'round to even'.

```
|- (round(X) To_nearest x =
      if x <= --(threshold(X)) then minus_infinity(X)
      else if x >= threshold(X) then plus_infinity(X)
      else closest (valof(X)) (λa. EVEN(fraction(a)))
           { a | is_finite(X) a} x)
```

[11] This assumption is later (6.1) specified in the standard, though it seems to be discussing the actual machine comparison operators rather than the mathematical framework in which the standard is interpreted.

We also define a function `intround` that rounds to an integer-valued floating point result. The definition is essentially the same as above, except that we restrict the set of floating point numbers to those that are finite *and* integer-valued, and, in the case of round-to-even, give preference to even integers, rather than merely integers with the least significant bit in their scaled representation zero. (For example, $2 = 2^1 \times 1.0000\ldots$ and $3 = 2^1 \times 1.1000\ldots$ both have their l.s.b. zero in the floating point representation.)

5.5 Arithmetic operations

The standard states (4.0) that each arithmetic operation (which excludes interconversion with decimal representations) is

> ...performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section.

This isn't quite the whole picture, since for example it doesn't settle the sign of zeros, and the standard later qualifies this by saying more precisely that an operation is:

> ...performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format (see Sections 4 and 7). Section 6 augments the following specifications to cover ± 0, $\pm\infty$ and NaN; Section 7 enumerates exceptions caused by exceptional operands and exceptional results.

Unfortunately section 7 gives another, more complicated, account of whether a number overflows when rounded (in the context of deciding whether to raise an exception):

> ...whenever the destination format's largest finite number is exceeded in magnitude by what would have been the correctly rounded floating point result (Section 4) were the exponent range unbounded.

However this is in fact equivalent to the definition in Section 4 that we formalized above, remembering the rounding to even restriction. Section 7 also fills out that description with a more explicit discussion of how overflow is dealt with in the various modes, which coincides with our formalization.

We depart from the standard slightly in our treatment of NaNs. The standard leaves many aspects of NaNs undetermined. It is stated that there are both 'quiet' and 'signalling' NaNs, and there are rules about propagating NaNs. However the division is an implementation issue, and to do justice to it, we would need to make the specification of arithmetic operations nondeterministic. Since IEEE arithmetic is completely determined on all other operands, this seems a pity. Therefore we just declare one canonical NaN and generate that in all situations where a NaN is required; we also make no distinction between quiet and signalling NaNs in the arguments. We do not discuss the raising of exceptions, nor of handlers for them.[12]

```
|- some_nan(X) = εa. is_nan(X) a
```

[12]Miner (1995) formalizes some of the rules for NaNs, but still relies on arbitrary but fixed choices for things not constrained by the standard, and does not discuss trap handlers. Barratt (1989) as far as we can tell gives a relational semantics that reflects the choices in the standard.

In other respects, the operations are fairly straightforward. One first deals with the exceptional cases, either where the arguments involve a NaN or infinity, or are invalid for other reasons (e.g. $\infty - \infty$). Apart from that, one basically just takes the values of the arguments, performs the mathematical operation on them, then rounds the result according to the desired rounding mode. The only additional problem is fixing the signs of zero results afterwards. We define the following function, which coerces the sign of a floating point number a with value zero to a given value s:

```
|- zerosign(X) s a = if is_zero(X) a then
                       if s = 0 then plus_zero(X) else minus_zero(X)
                     else a
```

Generally the rules for signs given in the standard (6.3) are quite simple. Note that in the case of sums and differences, the result is rounded to zero iff the precise result is zero, because the operation took place on representable numbers. So we only need to attend to zero results, and the rules are quite clear. For example:

```
|- fadd(X) m a b =
      if is_nan(X) a ∨ is_nan(X) b ∨
         (is_infinity(X) a ∧ is_infinity(X) b ∧ ¬(sign(a) = sign(b)))
      then some_nan(X)
      else if is_infinity(X) a then a
      else if is_infinity(X) b then b
      else zerosign(X) (if is_zero(X) a ∧ is_zero(X) b ∧
                           (sign(a) = sign(b)) then sign(a)
                        else if m = To_ninfinity then 1 else 0)
               (round(X) m (valof(X) a + valof(X) b))
```

and

```
|- fmul(X) m a b =
      if is_nan X a ∨ is_nan X b ∨
         is_zero X a ∧ is_infinity X b ∨
         is_infinity X a ∧ is_zero X b
      then some_nan X
      else if is_infinity X a ∨ is_infinity X b
           then if sign a = sign b
                then plus_infinity X
                else minus_infinity X
           else zerosign X (if sign a = sign b then 0 else 1)
                (round X m (valof X a * valof X b))
```

We also provide a negation operation on floating point numbers; this is recommended but not mandated by the IEEE standard. It simply switches the sign of a floating point number; apart from the signs of zeros it has the same effect as $-x = 0 - x$.

5.6 Comparisons

The standard allows two different ways of implementing the set of comparison operations: either as a function that returns a condition code, or as a set of standard predicates. Following Miner (1995), we do both, starting with an enumerated type of condition codes:

```
define_type "ccode = Gt | Lt | Eq | Un";;
```

The generic comparison function is rather obvious, though the case splits to deal with infinities are a bit tedious:

```
|- fcompare(X) a b =
      if is_nan(X) a ∨ is_nan(X) b then Un
      else if is_infinity(X) a ∧ (sign(a) = 1) then
        if is_infinity(X) b ∧ (sign(b) = 1) then Eq else Lt
      else if is_infinity(X) a ∧ (sign(a) = 0) then
        if is_infinity(X) b ∧ (sign(b) = 0) then Eq else Gt
      else if is_infinity(X) b ∧ (sign(b) = 1) then Gt
      else if is_infinity(X) b ∧ (sign(b) = 0) then Lt
      else if valof(X) a < valof(X) b then Lt
      else if valof(X) a = valof(X) b then Eq
      else Gt
```

Now we define the usual predicates, e.g.

```
|- flt(X) a b = (fcompare(X) a b = Lt)

|- fge(X) a b = (fcompare(X) a b = Gt) ∨ (fcompare(X) a b = Eq)
```

5.7 Float and double types

We specialize the above considerations to actual HOL types of single precision and double precision numbers (fixing also on round-to-nearest), called `float` and `double` following the C convention. (If HOL supported dependent types, we could actually have parametrized floating point *types*, as in Miner (1995).) These types are defined to be in bijection with the appropriate subset of $\mathbb{N}^3$, with the bijections for `float` and their types being written in HOL as `float:num#num#num->float` and `defloat:float->num#num#num`. The format of 8-bit exponents and 23-bit significands is abbreviated `float_format`:

```
|- float_format = (8,23)
```

The operations are defined by mapping out of the type, performing the operations, and mapping back, e.g.

```
|- a + b =
     float(fadd(float_format) To_nearest (defloat a) (defloat b))

|- sqrt(a) = float(fsqrt(float_format) To_nearest (defloat a))

|- a >= b = fge(float_format) (defloat a) (defloat b)
```

We also provide an 'absolute value' function as follows:[13]

```
|- abs a = if a >= Plus_zero then a else --a
```

The operations on floats 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.

[13] Note that we have $abs(-0) = -0$; it's probably more sensible to make $abs(-0) = +0$, but this decision does not affect the algorithm considered here.

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 $Val : \mathbb{F} \to \mathbb{R}$ (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 the operations, we specialize various other functions, predicates and values to the float type, e.g:

```
|- Val(a) = valof(float_format) (defloat a)

|- Float(x) = float (round(float_format) To_nearest x)

|- Isnan(a) = is_nan(float_format) (defloat a)

|- Plus_infinity = float (plus_infinity(float_format))
```

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. an arbitrary value $\varepsilon x.\ \bot$ 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.

6 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 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, 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)))
```

and

```
|- 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))
```

There are also many more trivial facts which nonetheless need to be established formally, e.g. that if a number is finite it is not a NaN and has a value in a certain range, that the equality is reflexive except on NaNs, that rounding is monotonic, and so forth, e.g.

```
|- ∀a. ¬(Isnan a ∧ Infinity a) ∧
       ¬(Isnan a ∧ Finite a) ∧
       ¬(Infinity a ∧ Finite a)

|- Iszero Plus_zero ∧ Iszero Minus_zero

|- ¬Isnan Plus_infinity ∧ ¬Isnan Minus_infinity

|- ¬Isnan Plus_zero ∧ ¬Isnan Minus_zero

|- ∀a. Infinity a = a == Plus_infinity ∨ a == Minus_infinity

|- ∀a. ¬(a == Plus_infinity ∧ a == Minus_infinity)
```

We will not catalogue every such result here but we should make clear that proving all these obvious (and sometimes not-so-obvious) facts took several days of work. Nevertheless, all the results described in this section are quite general, so could be re-used in any similar verifications undertaken in the future. A similar suite of trivialities needs to be provided for the machine integers and the operations relating them to floats, e.g.

```
|- Isintegral a ∧ --(&2 pow 31) <= Val a ∧ Val a < &2 pow 31
   ⟹ (Ival (Toint a) = Val a)
```

We use a separate valuation function `Ival` for machine integers; recall that `Val` is only used for floats.

7 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. The algorithm works as follows.

Given an input argument x, exceptional cases such as NaNs, infinities (or simply very large arguments) and zeros are dealt with first. For example, we have $exp(-\infty) = +0$. Furthermore, if the argument x is small enough for this to be a satisfactory approximation, the exponential function is calculated simply as $1 + x$. The main part of the algorithm covers the remaining cases. Mathematically, the procedure is simple. 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 $p(r) \approx e^r - 1$, where:

$$p(r) = r + \frac{8388676}{2^{24}} r^2 + \frac{11184876}{2^{26}} r^3$$

The actual reconstruction of e^x, for reasons of accuracy, is done by:

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

In fact, in order to achieve good accuracy, the above mathematical description is complicated slightly. The value r is broken down into $r_1 + r_2$ where $r_2 \ll r_1$. Similarly the prestored constants $2^{\frac{j}{32}}$ are all stored as two separate arrays S_{lead} and S_{trail} with $2^{\frac{j}{32}} \approx S_{lead}(j) + S_{trail}(j)$ and $S_{trail}(j) \ll S_{lead}(j)$. These devices to avoid rounding error, as well as the care required over the ordering of operations, make the actual code look a bit more complicated than the above mathematical description; they also create a lot of the difficulty in the verification. 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))
```

8 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, i.e. the inaccuracy compared with the ideal equation $X = N\frac{ln(2)}{32} + R$, where $R = R_1 + R_2$.
- The error resulting from the difference between $p(R)$ and $e^R - 1$, both regarded as pure mathematical expressions.
- The rounding error in the evaluation of $p(R)$ and the rest of the reconstruction.

We insert some annotations into the program which separate the correctness proof into three corresponding verification conditions (one of which splits into two because of the conditional statement), as well as half a dozen other VCs, most of which are trivial, for the special cases. A few additional assertions are added too, to break the task down further in a controlled way. For example, an assertion after the assignment to `E1` separates the detailed analysis of the rounding error and the final consideration of overflow and underflow on scaling by 2^M.

```
  ...
  E1 := S_Lead(J) + (S_Trail(J) + S * P);
  { Finite(X) ∧
    abs(Val X) <= Val(THRESHOLD_1) ∧
    Val(THRESHOLD_2) <= abs(Val X) ∧
    abs(Ival M) <= &318 ∧
    Finite(E1) ∧
    inv(&2) < Val(E1) ∧ Val(E1) < &2 - inv(&2 pow 6) ∧
    abs(Val(E1) - exp(Val(X) - Ival(M) * ln(&2)))
    <= &5338 / &10000 * inv (&2 pow 23) ∧
    (Val(E1) < &1 ⟹ abs(Val(E1) - exp(Val(X) - Ival(M) * ln(&2)))
                     <= &5125 / &10000 * inv (&2 pow 24)) };
  E := Scalb(E1,M)
  ...
```

In all, there are 13 verification conditions to be proved.

8.1 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) - &5445319802026545833408256866208 47 /
                &7855935874438170818322297257984 00) < 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.

8.2 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))
```

8.3 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 1997), 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.

8.4 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.[14] 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 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

[14] 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.

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.

8.5 Overflow

Our specification makes quite a strong statement about the overflow behaviour of the algorithm.[15] 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:

```
|- 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.

[15] 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.

8.6 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.

9 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. There has been other work in this area, which we will discuss briefly below, but ours is distinguished by a combination of factors:

- The algorithm is not a toy example, but a real-world, published one.
- The algorithm is for a transcendental function and the mathematics underlying it is non-trivial.
- The algorithm is expressed in a recognizable programming language.
- The whole algorithm is verified in full against a formal specification of IEEE-754.
- The entire proof proceeds from logical first principles.

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 1997) 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.[16] 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 1996c). 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 approaches. 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),

[16]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`.

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;[17] 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 1996c).

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.

[17] More precisely, to build HOL Light from the ground including the mathematical theories and then run through all the verification, takes 498 minutes of user CPU time on a 200MHz Pentium Pro machine with 128M of RAM running CAML Light 0.73 under Red Hat Linux 2.0.29 #10. Note that CAML Light is a bytecode interpreter, not a native code compiler.

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

Back, R. (1980) *Correctness Preserving Program Transformations: Proof Theory and Applications*, Volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam.

Baker, A. (1975) *Transcendental Number Theory*. Cambridge University Press.

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

Barwise, J. (1989) Mathematical proofs of computer correctness. *Notices of the American Mathematical Society*, **7**, 844–851.

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.

Clenshaw, C. W. and Olver, F. W. J. (1984) Beyond floating point. *Journal of the ACM*, **31**, 319–328.

DeMillo, R., Lipton, R., and Perlis, A. (1979) Social processes and proofs of theorems and programs. *Communications of the ACM*, **22**, 271–280.

Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice-Hall.

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

Gal, S. (1986) Computing elementary functions: A new approach for achieving high accuracy and good performance. In Miranker, W. L. and Toupin, R. A. (eds.), *Accurate scientific computations*, Volume 235 of *Lecture Notes in Computer Science*, pp. 1–16. Springer-Verlag.

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. (1995) The semantic challenge of Verilog HDL. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science*, San Diego, CA, USA, pp. 136–145. IEEE Computer Society Press.

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

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Harrison, J. (1995) Floating point verification in HOL. In Windley, P. J., Schubert, T., and Alves-Foss, J. (eds.), *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, Volume 971 of *Lecture Notes in Computer Science*, Aspen Grove, Utah, pp. 186–199. Springer-Verlag.

Harrison, J. (1996a) A Mizar mode for HOL. See von Wright, Grundy, and Harrison (1996), pp. 203–220.

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

Harrison, J. (1996c) Theorem proving with the real numbers. Technical Report 408, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK. Author's PhD thesis.

Harrison, J. (1997) 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.

IEEE (1985) Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

Lindemann, F. (1882) Über die Zahl π. *Mathematische Annalen*, **120**, 213–225.

Loveland, D. W. (1968) Mechanical theorem-proving by model elimination. *Journal of the ACM*, **15**, 236–251.

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`.

Ng, K. C. (1992) Argument reduction for huge arguments: Good to the last bit. Unpublished draft, available from the author (`kwok.ng@eng.sun.com`).

Nipkow, T. (1996) Winskel is (almost) right: Towards a mechanized semantics textbook. In Chandru, V. and Vinay, V. (eds.), *Foundations of Software Technology and Theoretical Computer Science, 16th conference, proceedings*, Lecture Notes in Computer Science, pp. 180–192. Springer-Verlag.

Norrish, M. (1997) An abstract dynamic semantics for C. Technical Report 421, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Page, I. (1996) Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, **12**, 87–107.

Paulson, L. C. (1987) *Logic and computation: interactive proof with Cambridge LCF*, Volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Payne, M. and Hanek, R. (1983) Radian reduction for trigonometric functions. *SIGNUM Newsletter*, **18**(1), 19–24.

Pollack, R. (1997) How to believe a machine-checked proof. In Sambin, G. and Smith, J. (eds.), *Twenty-Five Years of Constructive Type Theory*. Oxford University Press. To appear. Also available on the Web as `http://www.brics.dk/~pollack/export/BelieveMachineCheckedProof.ps.gz`.

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.

Trybulec, A. (1978) The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, **6**, 136–140.

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.

von Wright, J., Hekanaho, J., Luostarinen, P., and Långbacka, T. (1993) Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, **3**, 49–82.

Weis, P. and Leroy, X. (1993) *Le langage Caml*. InterEditions. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.

Wilkinson, J. H. (1963) *Rounding Errors in Algebraic Processes*, Volume 32 of *National Physical Laboratory Notes on Applied Science*. Her Majesty's Stationery Office (HMSO), London.