Floating-Point Verification using Theorem Proving

John Harrison

Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR 97124
`johnh@ichips.intel.com`

Abstract. This chapter describes our work on formal verification of floating-point algorithms using the HOL Light theorem prover.

1 Introduction

Representation of real numbers on the computer is fundamental to much of applied mathematics, from aircraft control systems to weather forecasting. Most applications use floating-point approximations, though this raises significant mathematical difficulties because of rounding and approximation errors. Even if rounding is properly controlled, "bugs" in software using real numbers can be particularly subtle and insidious. Yet because real-number programs are often used in controlling and monitoring physical systems, the consequences can be catastrophic. A spectacular example is the destruction of the Ariane 5 rocket shortly after takeoff in 1996, owing to an uncaught floating-point exception. Less dramatic, but very costly and embarrassing to Intel, was an error in the FDIV (floating-point division) instruction of some early Intel® Pentium® processors in 1994 [45]. Intel set aside approximately $475M to cover costs arising from this issue.

So it is not surprising that a considerable amount of effort has been applied to formal verification in the floating-point domain, not just at Intel [44, 34], but also at AMD [40, 50] and IBM [51], as well as in academia [33, 5]. Floating-point algorithms are in some ways an especially natural and appealing target for formal verification. It is not hard to come up with widely accepted formal specifications of how basic floating-point operations *should* behave. In fact, many operations are specified almost completely by the IEEE Standard governing binary floating-point arithmetic [32]. This gives a clear specification that high-level algorithms can rely on, and which implementors of instruction sets and compilers need to realize.

In some other respects though, floating-point operations present a difficult challenge for formal verification. In many other areas of verification, significant success has been achieved using highly automated techniques, usually based on a Boolean or other finite-state model of the state of the system. For example, efficient algorithms for propositional logic [7, 15, 53] and their aggressively efficient implementation [41, 19] have made possible a variety of techniques ranging from simple Boolean equivalence checking of combinational circuits to more advanced symbolic simulation or model checking of sequential systems [10, 47, 8, 52].

But it is less easy to verify non-trivial floating-point arithmetic operations using such techniques. The natural specifications, including the IEEE Standard, are based on real numbers, not bit-strings. While simple adders and multipliers can be specified quite naturally in Boolean terms, this becomes progressively more difficult when one considers division and square root, and seems quite impractical for transcendental functions. So while model checkers and similar tools are of great value in dealing with low-level details, at least some parts of the proof must be constructed in general theorem proving systems that enable one to talk about high-level mathematics.

There are many theorem proving programs,[1] and quite a few have been applied to floating-point verification, including at least ACL2, Coq, HOL Light and PVS. We will concentrate later on our own work using HOL Light [23], but this is not meant to disparage other important work being done at Intel and elsewhere in other systems.

2 Architectural context

Most of the general points we want to make here are independent of particular low-level details. Whether certain operations are implemented in software, microcode or hardware RTL, the general verification problems are similar, though the question of how they fit into the low-level design flow can be different. Nevertheless, in order to appreciate the examples that follow, the reader needs to know that the algorithms are mainly intended for software implementation on the Intel® Itanium® architecture, and needs to understand something about IEEE floating-point numbers and the special features of the Itanium floating-point architecture.

2.1 The IEEE-754 floating point standard

The IEEE Standard 754 for Binary Floating-Point Arithmetic [32] was developed in response to the wide range of incompatible and often surprising behaviours of floating-point arithmetic on different machines. In the 1970s, this was creating serious problems in porting numerical programs from one machine to another, with for example VAX, IBM/360 and Cray families all having different formats for floating point numbers and incompatible ways of doing arithmetic on them. Practically all of the arithmetics displayed peculiarities that could ensnare inexperienced programmers, for example $x = y$ being false but $x - y = 0$ true. The sudden proliferation of microprocessors in the 1970s promised to create even more problems, and was probably a motivating factor in the success of the IEEE standardization effort. Indeed, before the Standard was ratified, Intel had produced the 8087 math coprocessor to implement it, and other microprocessor makers followed suit. Since that time, practically every new microprocessor floating-point architecture has conformed to the standard.

Floating point numbers, at least in the conventional binary formats considered by the IEEE Standard 754,[2] are those of the form:

$$\pm 2^e k$$

---

[1] See `http://www.cs.ru.nl/~freek/digimath/index.html` for a list, and `http://www.cs.ru.nl/~freek/comparison/index.html` for a comparison of the formalization of an elementary mathematical theorem in several.

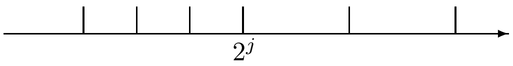
[2] The IBM/360 family, for example, used a hexadecimal not a binary system. Exotic formats such as those proposed in [11] depart even more radically from the Standard.

with the *exponent* e subject to a certain bound, and the *fraction* (also called significand or mantissa) k expressible in a binary positional representation using a certain number p of bits:

$$k = k_0 \cdot k_1 k_2 \cdots k_{p-1}$$

The bound on the exponent range $E_{min} \leq e \leq E_{max}$, together with the allowed *precision* p determines a particular floating point *format.* The novel aspect of the IEEE formats, and by far the most controversial part of the whole Standard, consists in allowing k_0 to be zero even when other k_i's are not. In the model established by most earlier arithmetics, the number $2^{E_{min}} 1.00 \cdots 0$ is the floating point number with smallest nonzero magnitude. But the IEEE-754 Standard allows for a set of smaller numbers $2^{E_{min}} 0.11 \cdots 11$, $2^{E_{min}} 0.11 \cdots 10$, $\ldots$, $2^{E_{min}} 0.00 \cdots 01$ which allow a more graceful underflow. Note that the above presentation enumerates some values redundantly, with for example $2^{e+1} 0 \cdot 100 \cdots 0$ and $2^e 1 \cdot 000 \cdots 0$ representing the same number. For many purposes, we consider the *canonical* representation, where $k_0 = 0$ only if $e = E_{min}$. Indeed, when it specifies how some floating point formats (single and double precision) are encoded as bit patterns, the Standard uses an encoding designed to avoid redundancy. Note, however, that the Standard specifies that the floating point representation must maintain the distinction between a positive and negative zero.

Floating point numbers cover a wide range of values from the very small to the very large. They are evenly spaced except that at the points 2^j the interval between adjacent numbers doubles. (Just as in decimal the gap between 1.00 and 1.01 is ten times the gap between 0.999 and 1.00, where all numbers are constrained to three significant digits.) The intervals $2^j \leq x \leq 2^{j+1}$, possibly excluding one or both endpoints, are often called *binades* (by analogy with 'decades'), and the numbers 2^j *binade boundaries.* The following diagram illustrates this.



In the IEEE Standard, the results of the basic algebraic operations (addition, subtraction, multiplication, division, remainder and square root) on finite operands[3] are specified in a simple and uniform way. According to the standard (section 5):

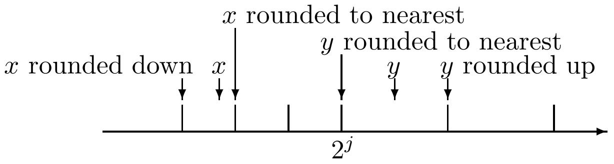
> . . . each of these operations shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range and then coerced this intermediate result to fit in the destination's format.

---

[3] In other words, those representing ordinary floating point numbers, rather than infinities or 'NaN's (NaN = not a number).

In non-exceptional situations (no overflow etc.) the coercion is done using a simple and natural form of *rounding*, defined in section 4 of the standard. Rounding essentially takes an arbitrary real number and returns another real number that is its best floating point approximation. Which particular number is considered 'best' may depend on the *rounding mode*. In the usual mode of *round to nearest*, the representable value closest to the exact one is chosen. If two representable values are equally close, i.e. the exact value is precisely the midpoint of two adjacent floating point numbers, then the one with its least significant bit zero is delivered. (See [48] for a problem caused by systematically truncating rather than rounding to even in such cases.)

Other rounding modes force the exact real number to be rounded to the representable number closest to it yet greater or equal ('round toward $+\infty$' or 'round up'), less or equal ('round toward $-\infty$' or 'round down') or smaller or equal in magnitude ('round toward 0'). The following diagram illustrates some IEEE-correct roundings; y is assumed to be exactly a midpoint, and rounds to 2^j because of the round-to-even preference.



2.2 The Intel® Itanium® floating point architecture

The Intel® Itanium® architecture is a 64-bit computer architecture jointly developed by Hewlett-Packard and Intel. In an attempt to avoid some of the limitations of traditional architectures, it incorporates a unique combination of features, including an instruction format encoding parallelism explicitly, instruction predication, and speculative/advanced loads [17]. However, we will not need to discuss general features like this, but only the special features of its floating-point architecture.

The centerpiece of the Intel® Itanium® floating-point architecture is the `fma` (floating point multiply-add or fused multiply-accumulate) instruction. This computes $xy + z$ from inputs x, y and z with a single rounding error. Except for subtleties over signed zeros, floating point addition and multiplication are just degenerate cases of `fma`, $1y + z$ and $xy + 0$, so do not need separate instructions. However, there are variants of the `fma` to switch signs of operands: `fms` computes $xy - z$ while `fnma` computes $z - xy$. While the IEEE standard does not explicitly address `fma`-type operations, the main extrapolation is obvious and natural: these operations behave as if they rounded an exact result, $xy + z$ for `fma`. The fact that there is only *one* rounding at the end, with no intermediate rounding of the product xy, is crucial in much of what follows.

It needs a little more care to specify the signs of zero results in IEEE style. First, the interpretation of addition and multiplication as degenerate cases of `fma` requires some policy on the sign of $1 \times -0 + 0$. More significantly, the `fma` leads to a new possibility: $a \times b + c$ can round to zero even though the exact result is nonzero. Out of the operations in the standard, this can occur for multiplication or division, but in this case the rules for signs are simple and natural. A little reflection shows that this cannot happen for pure addition, so the rule in the standard that 'the sign of a sum ... differs from at most one of the addend's signs' is enough to fix the sign of zeros when the exact result is nonzero. For the `fma` this is not the case, and `fma`-type instructions guarantee that the sign correctly reflects the sign of the exact result in such cases. This is important, for example, in ensuring that the algorithms for division that we consider later yield the correctly signed zeros in all cases without special measures.

The Intel® Itanium® architecture supports several different IEEE-specified or IEEE-compatible floating point formats. For the four most important formats, we give the conventional name, the precision, and the minimum and maximum exponents.

Format name	p	E_{min}	E_{max}
Single	24	-126	127
Double	53	-1022	1023
Double-extended	64	-16382	16383
Register	64	-65534	65535

The single and double formats are mandated and completely specified in the Standard. The double-extended format (we will often just call it 'extended') is recommended and only partially specified by the Standard; the particular version used in Intel® Itanium® is the one introduced by on the 8087.[4] The register format has the same precision as extended, but allows greater exponent range, helping to avoid overflows and underflows in intermediate calculations. In a sense, the register format is all-inclusive, since its representable values properly include those of all other formats.

Most operations, including the `fma`, take arguments and return results in some of the 128 floating point registers provided for by the architecture. All the formats are mapped into a standard bit encoding in registers, and any value represented by a floating point register is representable in the 'Register' floating point format.[5] By a combination of settings in the multiple status fields and completers on instructions, the results of operations can be rounded in any of the four rounding modes and into any of the supported formats.

In most current computer architectures, in particular Intel IA-32 (x86) currently represented by the Intel® Pentium® processor family, instructions are specified for the floating point division and square root operations. In the Itanium architecture, the only instructions specifically intended to support division and square root are initial approximation instructions. The `frcpa` (floating

---

[4] A similar format was supported by the Motorola 68000 family, but now it is mainly supported by Intel processors.

[5] However, that for reasons of compatibility, there are bit encodings used for extended precision numbers but not register format numbers.

point reciprocal approximation) instruction applied to an argument a gives an approximation to $1/a$, while the `frsqrta` (floating point reciprocal square root approximation) instruction gives an approximation to $1/\sqrt{a}$. (In each case, the approximation may have a relative error of approximately $2^{-8.8}$ in the worst case, so they are far from delivering an exact result. Of course, special action is taken in some special cases like $a = 0$.) There are several reasons for relegating division and square root to software.

- In typical applications, division and square root are not extremely frequent operations, and so it may be that die area on the chip would be better devoted to something else. However they are not so infrequent that a grossly inefficient software solution is acceptable, so the rest of the architecture needs to be designed to allow reasonably fast software implementations. As we shall see, the `fma` is the key ingredient.
- By implementing division and square root in software they immediately inherit the high degree of pipelining in the basic `fma` operations. Even though these operations take several clock cycles, new ones can be started each cycle while others are in progress. Hence, many division or square root operations can proceed in parallel, leading to much higher throughput than is the case with typical hardware implementations.
- Greater flexibility is afforded because alternative algorithms can be substituted where it is advantageous. First of all, any improvements can quickly be incorporated into a computer system without hardware changes. Second, it is often the case that in a particular context a faster algorithm suffices, e.g. because the ambient IEEE rounding mode is known at compile-time, or even because only a moderately accurate result is required (e.g. in some graphics applications).

3 HOL Light

Theorem provers descended from Edinburgh LCF [22] reduce all reasoning to formal proofs in something like a standard natural deduction system. That is, everything must be proved in detail at a low level, not simply asserted, nor even claimed on the basis of running some complex decision procedure. However, the user is able to write arbitrary programs in the metalanguage ML to automate patterns of inferences and hence reduce the tedium involved. The original LCF system implemented Scott's Logic of Computable Functions (hence the name LCF), but as emphasized by Gordon [20], the basic LCF approach is applicable to any logic, and now there are descendents implementing a variety of higher order logics, set theories and constructive type theories.

In particular, members of the HOL family [21] implement a version of simply typed λ-calculus with logical operations defined on top, more or less following Church [9]. They take the LCF approach a step further in that all theory developments are pursued 'definitionally'. New mathematical structures, such as the real numbers, may be defined only by exhibiting a model for them in the existing theories (say as Dedekind cuts of rationals). New constants may only be introduced by definitional extension (roughly speaking, merely being a shorthand for an expression in the existing theory). This fits naturally with the LCF style, since it ensures that all extensions, whether of the deductive system or the mathematical theories, are consistent per construction. HOL Light [23] is our own version of the HOL prover. It maintains most of the general principles underlying its ancestors, but attempts to be more logically coherent, simple and elegant. It is written entirely in a fairly simple and mostly functional subset of Objective CAML [57, 14], giving it advantages of portability and efficient resource usage compared with its ancestors, which are based on LISP or Standard ML.

Like other LCF provers, HOL Light is in essence simply a large OCaml 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 HOL's 10 very simple inference rules or to make a new term or type definition. 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 OCaml's type checking etc.). For example, one of HOL's primitives is the rule of transitivity of equality:

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

This allows one to make the following logical step: if under assumptions Γ one can deduce $s = t$ (that is, s and t are equal), and under assumptions Δ one can deduce $t = u$, then one can deduce that from all the assumptions together, $\Gamma \cup \Delta$, that $s = u$ 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 = TRANS 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 read and 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.[6] 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 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, in HOL's inbuilt foundations of number theory, the proof 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 [36] 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[]);;
```

---

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

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, OCaml programs, albeit of a fairly stylized form. Perhaps in the future a more declarative style for proof scripts will prove to be more effective, but the procedural approach has its merits [24], particularly in applications like this one where we program several complex specialized inference rules.

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
$\bot$	`F`	falsity
$\top$	`T`	truth
$\neg p$	`~p`	not p
$p \land q$	`p /\ q`	p and q
$p \lor q$	`p \/ q`	p or q
$p \Longrightarrow q$	`p ==> q`	if p then q
$p \Leftrightarrow q$	`p <=> q`	p if and only if q
$\forall x.\, p$	`!x. p`	for all x, p
$\exists x.\, p$	`?x. p`	there exists x such that p
$\varepsilon x.\, p$	`@x. p`	some x such that p
$\lambda x.\, t$	`\x. t`	the function $x \mapsto t$
$x \in s$	`x IN s`	x is a member of set s

For more on the fundamentals of the HOL logic, see the appendix of our PhD dissertation [25], or actually download the system and its documentation from:

`http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`

Our thesis [25] also gives extensive detail on the definition of real numbers in HOL and the formalization of mathematical analysis built on this foundation. In what follows, we exploit various results of pure mathematics without particular comment. However, it may be worth noting some of the less obvious aspects of the HOL symbolism when dealing with numbers:

HOL notation	Standard symbol	Meaning
`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

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.

4 Application examples

We will now give a brief overview of some of our verification projects using HOL Light at Intel. For more details on the various parts, see [26, 28, 29, 27].

4.1 Formalization of floating point arithmetic

The first stage in all proofs of this kind is to formalize the key floating point concepts and prove the necessary general lemmas. We have tried to provide a generic, re-usable theory of useful results that can be applied conveniently in these and other proofs. In what follows we will just discuss things in sufficient detail to allow the reader to get the gist of what follows.

Floating point numbers can be stored either in floating point registers or in memory, and in each case we cannot always assume the encoding is irredundant (i.e. there may be several different encodings of the same real value, even apart from IEEE signed zeros). Thus, we need to take particular care over the distinction between values and their floating point encodings.[7] Systematically making this separation nicely divides our formalization into two parts: those that are concerned only with real numbers, and those where the floating point encodings with the associated plethora of special cases (infinities, NaNs, signed zeros etc.) come into play. Most of the interesting issues can be considered at the level of real numbers, and we will generally follow this approach here. However there are certainly subtleties over the actual representations that we need to be aware of, e.g. checking that zero results have the right sign and characterizing the situations where exceptions can or will occur.

Floating point formats Our formalization of the encoding-free parts of the standard is highly generic, covering an infinite collection of possible floating point formats, even including absurd formats with zero precision (no fraction bits). It

---

[7] In the actual standard (p7) 'a bit-string is not always distinguished from a number it may represent'.

is a matter of taste whether the pathological cases should be excluded at the outset. We sometimes need to exclude them from particular theorems, but many of the theorems turn out to be degenerately true even for extreme values.

Section 3.1 of the standard parametrizes floating point formats by precision p and maximum and minimum exponents E_{max} and E_{min}. We follow this closely, except we represent the fraction by an integer rather than a value $1 \leq f < 2$, and the exponent range by two nonnegative numbers N and E. The allowable floating point numbers are then of the form $\pm 2^{e-N}k$ with $k < 2^p$ and $0 \leq e < E$. This was not done because of the use of biasing in actual floating point encodings (as we have stressed before, we avoid such issues at this stage), but rather to use nonnegative integers everywhere and carry around fewer side-conditions. The cost of this is that one needs to remember the bias when considering the exponents of floating point numbers. We name the fields of a format triple as follows:

```
|- exprange (E,p,N) = E

|- precision (E,p,N) = p

|- ulpscale (E,p,N) = N
```

and the definition of the set of real numbers corresponding to a triple is:[8]

```
|- format (E,p,N) =
    { x | ∃s e k. s < 2 ∧ e < E ∧ k < 2 EXP p ∧
                  x = --(&1) pow s * &2 pow e * &k / &2 pow N}
```

This says exactly that the format is the set of real numbers representable in the form $(-1)^s 2^{e-N}k$ with $e < E$ and $k < 2^p$ (the additional restriction $s < 2$ is just a convenience). For many purposes, including floating point rounding, we also consider an analogous format with an exponent range unbounded above. This is defined by simply dropping the exponent restriction $e < E$. Note that the exponent is still bounded *below*, i.e. N is unchanged.

```
|- iformat (E,p,N) =
    { x | ∃s e k. s < 2 ∧ k < 2 EXP p ∧
                  x = --(&1) pow s * &2 pow e * &k / &2 pow N}
```

Ulps The term 'unit in the last place' (ulp) is only mentioned in passing by the standard on p. 12 when discussing binary to decimal conversion. Nevertheless, it is of great importance for later proofs because the error bounds for transcendental functions need to be expressed in terms of ulps. Doing so is quite standard,

---

[8] Recall that the ampersand denotes the injection from $\mathbb{N}$ to $\mathbb{R}$, which HOL's type system distinguishes. The function `EXP` denotes exponentiation on naturals, and `pow` the analogous function on reals.

yet there is widespread confusion about what an ulp is, and a variety of incompatible definitions appear in the literature [43]. Given a particular floating-point format and a real number x, we define an ulp in x as the distance between the two closest *straddling* floating point numbers a and b, i.e. those with $a \leq x \leq b$ and $a \neq b$ assuming an unbounded exponent range.

This seems to convey the natural intuition of units in the last place, and preserves the important mathematical properties that rounding to nearest corresponds to an error of $0.5ulp$ and directed roundings imply a maximum error of $1ulp$. The actual HOL definition is explicitly in terms of binades, and defined using the Hilbert choice operator ε:[9]

```
|- binade(E,p,N) x =
      εe. abs(x) <= &2 pow (e + p) / &2 pow N ∧
          ∀e'. abs(x) <= &2 pow (e' + p) / &2 pow N ⟹ e <= e'

|- ulp(E,p,N) x = &2 pow (binade(E,p,N) x) / &2 pow N
```

After a fairly tedious series of proofs, we eventually derive the theorem that an ulp does indeed yield the distance between two straddling floating point numbers.

Rounding Floating point rounding takes an arbitrary real number and chooses a floating point approximation. Rounding is regarded in the Standard as an operation mapping a real to a member of the extended real line $\mathbb{R} \cup \{+\infty, -\infty\}$, not the space of floating point numbers itself. Thus, encoding and representational issues (e.g. zero signs) are not relevant to rounding. The Standard defines four rounding modes, which we formalize as the members of an enumerated type:

```
roundmode = Nearest | Down | Up | Zero
```

Our formalization defines rounding into a given format as an operation that maps into the corresponding format *with an exponent range unbounded above.* That is, we do not take any special measures like coercing overflows back into the format or to additional 'infinite' elements; this is defined separately when we consider operations. While this separation is not quite faithful to the letter of the Standard, we consider our approach preferable. It has obvious technical convenience, avoiding the formally laborious adjunction of infinite elements to the real line and messy side-conditions in some theorems about rounding. Moreover, it avoids duplication of closely related issues in different parts of the Standard. For example, the rather involved criterion for rounding to $\pm\infty$ in round-to-nearest mode in sec. 4.1 of the Standard ('an infinitely precise result with magnitude at least $E_{max}(2 - 2^{-p})$ shall round to ∞ with no change of sign') is not needed. In our setup we later consider numbers that round to values outside the range-restricted format as overflowing, so the exact same condition is implied. This

---

[9] Read 'ε`e. ...`' as 'the `e` such that ...'.

approach in any case *is* used later in the Standard 7.3 when discussing the raising of the overflow exception ('...were the exponent range unbounded').

Rounding is defined in HOL as a direct transcription of the Standard's definition. There is one clause for each of the four rounding modes:

```
|- (round fmt Nearest x =
      closest_such (iformat fmt) (EVEN o decode_fraction fmt) x) ∧
   (round fmt Down x = closest {a | a ∈ iformat fmt ∧ a <= x} x) ∧
   (round fmt Up x = closest {a | a ∈ iformat fmt ∧ a >= x} x) ∧
   (round fmt Zero x =
      closest {a | a ∈ iformat fmt ∧ abs a <= abs x} x)
```

For example, the result of rounding `x` down is defined to be the closest to `x` of the set of real numbers `a` representable in the format concerned (`a ∈ iformat fmt`) and no larger than `x` (`a <= x`). The subsidiary notion of 'the closest member of a set of real numbers' is defined using the Hilbert ε operator. As can be seen from the definition, rounding to nearest uses a slightly elaborated notion of closeness where the result with an even fraction is preferred.[10]

```
|- is_closest s x a ⇔
      a ∈ s ∧ ∀b. b ∈ s ⟹ abs(b - x) >= abs(a - x)

|- closest s x = εa. is_closest s x a

|- closest_such s p x =
     εa. is_closest s x a ∧ (∀b. is_closest s x b ∧ p b ⟹ p a)
```

In order to derive useful consequences from the definition, we then need to show that the postulated closest elements always exist. Actually, this depends on the format's being nontrivial. For example, if the format has nonzero precision, then rounding up behaves as expected:

```
|- ¬(precision fmt = 0)
   ⟹ round fmt Up x ∈ iformat fmt ∧
      x <= round fmt Up x ∧
      abs(x - round fmt Up x) < ulp fmt x ∧
      ∀c. c ∈ iformat fmt ∧ x <= c
          ⟹ abs(x - round fmt Up x) <= abs(x - c)
```

The strongest results for rounding to nearest depend on the precision being at least 2. This is because in a format with $p = 1$ nonzero normalized numbers all have fraction 1, so 'rounding to even' no longer discriminates between adjacent floating point numbers in the same way.

---

[10] Note again the important distinction between real values and encodings. The canonical fraction is used; the question of whether the actual floating point value has an even fraction is irrelevant. We do not show all the details of how the canonical fraction is defined.

The $(1+\epsilon)$ lemma The most widely used lemma about floating point arithmetic, often called the '$(1+\epsilon)$' property, is simply that the result of a floating point operation is the exact result, perturbed by a relative error of bounded magnitude. Recalling that in our IEEE arithmetic, the result of an operation is the rounded exact value, this amounts to saying that x rounded is always of the form $x(1+\epsilon)$ with $|\epsilon|$ bounded by a known value, typically 2^{-p} where p is the precision of the floating point format. We can derive a result of this form fairly easily, though we need sideconditions to exclude the possibility of underflow (not overflow, which we consider separately from rounding). The main theorem is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ⟹ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
           round fmt rc x = x * (&1 + e)
```

This essentially states exactly the '$1+e$' property, and the bound on ϵ depends on the rounding mode, according to the following auxiliary definition of `mu`:

```
|- mu Nearest = &1 / &2 ∧
   mu Down = &1 ∧
   mu Up = &1 ∧
   mu Zero = &1
```

The theorem has two sideconditions, the second being the usual nontriviality hypothesis, and the first being an assertion that the value x does not *lose precision*, in other words, that the result of rounding x would not change if the lower exponent range were extended. We will not show the formal definition [26] here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range (or is zero):

```
|- normalizes fmt x ⟹ ¬(losing fmt rc x)
```

where

```
|- normalizes fmt x ⇔
        x = &0 ∨
        &2 pow (precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(x)
```

In a couple of places, however, we need a sharper criterion for when the result of an `fma` operation will not lose precision. The proof of the following result merely observes that either the result is in the normalized range, or else the result will cancel so completely that the result will be exactly representable; however the technical details are non-trivial.

```
|- ¬(precision fmt = 0) ∧
   a ∈ iformat fmt ∧
   b ∈ iformat fmt ∧
   c ∈ iformat fmt ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(c)
   ⟹ ¬(losing fmt rc (a * b + c))
```

Lemmas about exactness The '$(1+\epsilon)$' property allows us to ignore most of the technical details of floating point rounding, and step back into the world of exact real numbers and straightforward algebraic calculations. Many highly successful backward error analyses of higher-level algorithms [59] often rely essentially only on this property. Indeed, for most of the operations in the algorithms we are concerned with here, '$(1 + \epsilon)$' is the only property needed for us to verify what we require of them.

However, in lower-level algorithms like the ones considered here and others that the present author is concerned with verifying, a number of additional properties of floating point arithmetic are sometimes exploited by the algorithm designer, and proofs of them are required for verifications. In particular, there are important situations where floating point arithmetic is exact, i.e. results round to themselves. This happens if and only if the result is representable as a floating point number:

```
|- a ∈ iformat fmt ⟹ round fmt rc a = a

|- ¬(precision fmt = 0) ⟹ (round fmt rc x = x ⇔ x ∈ iformat fmt)
```

There are a number of situations where arithmetic operations are exact. Perhaps the best-known instance is subtraction of nearby quantities; cf. Theorem 4.3.1 of [54]:

```
|- a ∈ iformat fmt ∧ b ∈ iformat fmt ∧ a / &2 <= b ∧ b <= &2 * a
   ⟹ (b - a) ∈ iformat fmt
```

The availability of an `fma` operation leads us to consider generalization where results with higher intermediate precision are subtracted. The following is a direct generalization of the previous theorem, which corresponds to the case $k = 0$.[11]

```
|- ¬(p = 0) ∧
   a ∈ iformat (E1,p+k,N) ∧
   b ∈ iformat (E1,p+k,N) ∧
   abs(b - a) <= abs(b) / &2 pow (k + 1)
   ⟹ (b - a) ∈ iformat (E2,p,N)
```

Another classic result [39, 16] shows that we can obtain the sum of two floating point numbers exactly in two parts, one a rounding error in the other, by performing the floating point addition then subtracting both summands from the result, the larger one first:

```
|- x ∈ iformat fmt ∧
   y ∈ iformat fmt ∧
   abs(x) <= abs(y)
   ⟹ (round fmt Nearest (x + y) - y) ∈ iformat fmt ∧
       (round fmt Nearest (x + y) - (x + y)) ∈ iformat fmt
```

---

[11] With an assumption that `a` and `b` belong to the same binade, the `(k + 1)` can be strengthened to `k`.

Once again, we have devised a more general form of this theorem, with the above being the $k = 0$ case.[12] It allows a laxer relationship between x and y if the smaller number has k fewer significant digits:

```
|- k <= ulpscale fmt ∧
   x ∈ iformat fmt ∧
   y ∈ iformat(exprange fmt,precision fmt - k,ulpscale fmt - k) ∧
   abs(x) <= abs(&2 pow k * y)
   ⟹ (round fmt Nearest (x + y) - y) ∈ iformat fmt ∧
       (round fmt Nearest (x + y) - (x + y)) ∈ iformat fmt
```

The `fma` leads to a new result of this kind, allowing us to obtain an exact product in two parts:[13]

```
|- a ∈ iformat fmt ∧ b ∈ iformat fmt ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a * b)
   ⟹ (a * b - round fmt Nearest (a * b)) ∈ iformat fmt
```

A note on floating point axiomatics The proofs of many of the above lemmas, and other more specialized results used in proofs, are often quite awkward and technical. In particular, they often require us to return to the basic definitions of floating point numbers and expand them out as sign, exponent and fraction before proceeding with some detailed bitwise or arithmetical proofs. This state of affairs compares badly with the organization of material in much of pure mathematics. For example the HOL theory of abstract real numbers proves from the definitions of reals (via a variant of Cauchy sequences) some basic 'axioms' from which all of real analysis is developed; once the reals have been constructed, the details of how the real numbers were constructed are irrelevant.

One naturally wonders if there is a clean set of 'axioms' from which most interesting floating point results can be derived more transparently. The idea of encapsulating floating point arithmetic in a set of axioms has attracted some attention over the years. However, the main point of these exercises was not so much to simplify and unify proofs in a *particular* arithmetic, but rather to produce proofs that would cover various different arithmetic implementations. With the almost exclusive use of IEEE arithmetic these days, that motivation has lost much of its force. Nevertheless it is worth looking at whether the axiom sets that have been proposed would be useful for our purposes.

The cleanest axiom systems are those that don't make reference to the underlying floating point representation [58, 60, 31]. However these are also the least useful for deriving results of the kind we consider, where details of the floating point representation are clearly significant. Moreover, at least one axiom in [60] is actually false for IEEE arithmetic, and according to [46], for almost every commercially significant machine except for the Cray X-MP and Y-MP.

---

[12] See [35] for other generalizations that we do not consider.

[13] We are not sure where this result originated; it appears in some of Kahan's lecture notes at `http://www.cs.berkeley.edu~wkahan/ieee754status/ieee754.ps`.

Generally speaking, in order to make it possible to prove the kind of subtle exactness results that we sometimes need, axioms that are explicit about the floating point representation are needed [16, 6]. Once this step is taken, there don't seem to be many benefits over our own explicit proofs, given that one is working in a precisely defined arithmetic. On the other hand, it is certainly interesting to see how many results can be proved by weaker hypotheses. For example, using the floating point representation explicitly but only assuming about rounding that it is, in Dekker's terminology, *faithful*, Priest [46] deduces several interesting consequences including Sterbenz's theorem about cancellation in subtraction, part of the theorem on exact sums, and the following:[14]

$$0 \leq a \leq b \wedge fl(b-a) = b-a \Longrightarrow \forall c.\, a \leq c \leq b \Longrightarrow fl(c-a) = c-a$$

However, he also makes some interesting remarks on the limitations of this axiomatic approach.

4.2 Division

It is not immediately obvious that without tricky and time-consuming bit-twiddling, it is possible to produce an IEEE-correct quotient and set all the IEEE flags correctly via ordinary software. Remarkably, however, fairly short straight-line sequences of `fma` operations (or negated variants), suffice to do so. This approach to division was pioneered by Markstein [38] on the IBM RS/6000[15] family. It seems that the ability to perform both a multiply and an add or subtract without an intermediate rounding is essential here.

Refining approximations First we will describe in general terms how we can use `fma` operations to refine an initial reciprocal approximation (obtained from `frcpa`) towards a better reciprocal or quotient approximation. For clarity of exposition, we will ignore rounding errors at this stage, and later show how they are taken account of in the formal proof.

Consider determining the reciprocal of some floating point value b. Starting with a reciprocal approximation y with a relative error ϵ:

$$y = \frac{1}{b}(1+\epsilon)$$

we can perform just one `fnma` operation:

$$e = 1 - by$$

and get:

---

[14] Think of fl as denoting rounding to nearest.

[15] All other trademarks are the property of their respective owners.

$$
\begin{aligned}
e &= 1 - by \\
&= 1 - b\frac{1}{b}(1+\epsilon) \\
&= 1 - (1+\epsilon) \\
&= -\epsilon
\end{aligned}
$$

Now observe that:

$$
\begin{aligned}
\frac{1}{b} &= \frac{y}{(1+\epsilon)} \\
&= y(1 - \epsilon + \epsilon^2 - \epsilon^3 + \cdots) \\
&= y(1 + e + e^2 + e^3 + \cdots)
\end{aligned}
$$

This suggests that we might improve our reciprocal approximation by multiplying y by some truncation of the series $1 + e + e^2 + e^3 + \cdots$. The simplest case using a linear polynomial in e can be done with just one more `fma` operation:

$$y' = y + ey$$

Now we have

$$
\begin{aligned}
y' &= y(1+e) \\
&= \frac{1}{b}(1+\epsilon)(1+e) \\
&= \frac{1}{b}(1+\epsilon)(1-\epsilon) \\
&= \frac{1}{b}(1-\epsilon^2)
\end{aligned}
$$

The magnitude of the relative error has thus been squared, or looked at another way, the number of significant bits has been approximately doubled. This, in fact, is exactly a step of the traditional Newton-Raphson iteration for reciprocals. In order to get a still better approximation, one can either use a longer polynomial in e, or repeat the Newton-Raphson linear correction several times. Mathematically speaking, repeating Newton-Raphson iteration n times is equivalent to using a polynomial $1 + e + \cdots + e^{2^n - 1}$, e.g. since $e' = \epsilon^2 = e^2$, two iterations yield:

$$y'' = y(1+e)(1+e^2) = y(1 + e + e^2 + e^3)$$

However, whether repeated Newton iteration or a more direct power series evaluation is better depends on a careful analysis of efficiency and the impact of rounding error. The Intel algorithms use both, as appropriate.

Now consider refining an approximation to the quotient with relative error ϵ; we can get such an approximation in the first case by simply multiplying a reciprocal approximation $y \approx \frac{1}{b}$ by a. One approach is simply to refine y as much as possible and then multiply. However, this kind of approach can never guarantee getting the last bit right; instead we also need to consider how to refine q directly. Suppose

$$q = \frac{a}{b}(1 + \epsilon)$$

We can similarly arrive at a remainder term by an `fnma`:

$$r = a - bq$$

when we have:

$$\begin{aligned} r &= a - bq \\ &= a - b\frac{a}{b}(1 + \epsilon) \\ &= a - a(1 + \epsilon) \\ &= -a\epsilon \end{aligned}$$

In order to use this remainder term to improve q, we also need a reciprocal approximation $y = \frac{1}{b}(1 + \eta)$. Now the `fma` operation:

$$q' = q + ry$$

results in, ignoring the rounding:

$$\begin{aligned} q' &= q + ry \\ &= \frac{a}{b}(1 + \epsilon) - a\epsilon\frac{1}{b}(1 + \eta) \\ &= \frac{a}{b}(1 + \epsilon - \epsilon(1 + \eta)) \\ &= \frac{a}{b}(1 - \epsilon\eta) \end{aligned}$$

Obtaining the final result While we have neglected rounding errors hitherto, it is fairly straightforward to place a sensible bound on their effect. To be precise, the error from rounding is at most half an ulp in round-to-nearest mode and a full ulp in the other modes.

```
⊢  ¬(precision fmt = 0)
   ⟹ (abs(error fmt Nearest x) <= ulp fmt x / &2) ∧
       (abs(error fmt Down x) < ulp fmt x) ∧
       (abs(error fmt Up x) < ulp fmt x) ∧
       (abs(error fmt Zero x) < ulp fmt x)
```

where

```
⊢  error fmt rc x = round fmt rc x - x
```

It turn, we can easily get fairly tight lower ($|x|/2^p \leq ulp(x)$) and upper ($ulp(x) \leq |x|/2^{p-1}$) bounds on an ulp in x relative to the magnitude of x, the upper bound assuming normalization:

```
⊢  abs(x) / &2 pow (precision fmt) <= ulp fmt x
```

and

```
⊢  normalizes fmt x ∧ ¬(precision fmt = 0) ∧ ¬(x = &0)
   ⟹ ulp fmt x <= abs(x) / &2 pow (precision fmt - 1)
```

Putting these together, we can easily prove simple relative error bounds on all the basic operations, which can be propagated through multiple calculations by simple algebra. It is easy to see that while the relative errors in the approximations are significantly above 2^{-p} (where p is the precision of the floating point format), the effects of rounding error on the overall error are minor. However, once we get close to having a perfectly rounded result, rounding error becomes highly significant. A crucial theorem here is the following due to Markstein [38]:

Theorem 1. *If q is a floating point number within 1 ulp of the true quotient a/b of two floating point numbers, and y is the correctly rounded-to-nearest approximation of the exact reciprocal $\frac{1}{b}$, then the following two floating point operations:*

$$
\begin{aligned}
r &= a - bq \\
q' &= q + ry
\end{aligned}
$$

using round-to-nearest in each case, yield the correctly rounded-to-nearest quotient q'.

This is not too difficult to prove in HOL. First we observe that because the initial q is a good approximation, the computation of r cancels so much that no rounding error is committed. (This is intuitively plausible and stated by Markstein without proof, but the formal proof was surprisingly messy.)

```
⊢  2 <= precision fmt ∧
   a ∈ iformat fmt ∧ b ∈ iformat fmt ∧ q ∈ iformat fmt ∧
   normalizes fmt q ∧ abs(a / b - q) <= ulp fmt (a / b) ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a)
   ⟹ (a - b * q) ∈ iformat fmt
```

Now the overall proof given by Markstein is quite easily formalized. However, we observed that the property actually used in the proof is in general somewhat weaker than requiring y to be a perfectly rounded reciprocal. The theorem actually proved in HOL is:

Theorem 2. *If* q *is a floating point number within* 1 *ulp of the true quotient* a/b *of two floating point numbers, and* y *approximates the exact reciprocal* $\frac{1}{b}$ *to a relative error* $< \frac{1}{2^p}$*, where* p *is the precision of the floating point format concerned, then the following two floating point operations:*

$$r = a - bq$$
$$q' = q + ry$$

using round-to-nearest in each case, yield the correctly rounded-to-nearest quotient q'*.*

The formal HOL statement is as follows:

```
⊢  2 <= precision fmt ∧
   a ∈ iformat fmt ∧ b ∈ iformat fmt ∧
   q ∈ iformat fmt ∧ r ∈ iformat fmt ∧
   ¬(b = &0) ∧
   ¬(a / b ∈ iformat fmt) ∧
   normalizes fmt (a / b) ∧
   abs(a / b - q) <= ulp fmt (a / b) ∧
   abs(inv(b) - y) < abs(inv b) / &2 pow (precision fmt) ∧
   r = a - b * q ∧
   q' = q + r * y
   ⟹ round fmt Nearest q' = round fmt Nearest (a / b)
```

Although in the worst case, the preconditions of the original and modified theorem hardly differ (recall that $|x|/2^p \leq ulp(x) \leq |x|/2^{p-1}$), it turns out that in many situations the relative error condition is much easier to satisfy. In Markstein's original methodology, one needs first to obtain a perfectly rounded reciprocal, which he proves can be done as follows:

Theorem 3. *If* y *is a floating point number within* 1 *ulp of the true reciprocal* $\frac{1}{b}$*, then one iteration of:*

$$e = 1 - by$$
$$y' = y + ey$$

using round-to-nearest in both cases, yields the correctly rounded reciprocal, except possibly when the mantissa of b *consists entirely of* $1s$*.*

If we rely on this theorem, we need a very good approximation to $\frac{1}{b}$ before these two further serial operations and one more to get the final quotient using the new y'. However, with the weaker requirement on y', we can get away with a correspondingly weaker y. In fact, we prove:

Theorem 4. *If y is a floating point number that results from rounding a value y_0, and the relative error in y_0 w.r.t. $\frac{1}{b}$ is $\leq \frac{d}{2^{2p}}$ for some natural number d (assumed $\leq 2^{p-2}$), then y will have relative error $< \frac{1}{2^p}$ w.r.t. $\frac{1}{b}$, except possibly if the mantissa of b is one of the d largest. (That is, when scaled up to an integer $2^{p-1} \leq m_b < 2^p$, we have in fact $2^p - d \leq m_b < 2^p$.)*

Proof. For simplicity we assume $b > 0$, since the general case can be deduced by symmetry from this. We can therefore write $b = 2^e m_b$ for some integer m_b with $2^{p-1} \leq m_b < 2^p$. In fact, it is convenient to assume that $2^{p-1} < m_b$, since when b is an exact power of 2 the main result follows easily from $d \leq 2^{p-2}$. Now we have:

$$\begin{aligned}\frac{1}{b} &= 2^{-e}\frac{1}{m_b}\\ &= 2^{-(e+2p-1)}(\frac{2^{2p-1}}{m_b})\end{aligned}$$

and $ulp(\frac{1}{b}) = 2^{-(e+2p-1)}$. In order to ensure that $|y - \frac{1}{b}| < |\frac{1}{b}|/2^p$ it suffices, since $|y - y_0| \leq ulp(\frac{1}{b})/2$, to have:

$$\begin{aligned}|y_0 - \frac{1}{b}| &< (\frac{1}{b})/2^p - ulp(\frac{1}{b})/2\\ &= (\frac{1}{b})/2^p - 2^{-(e+2p-1)}/2\\ &= (\frac{1}{b})/2^p - (\frac{1}{b})m_b/2^{2p}\end{aligned}$$

By hypothesis, we have $|y_0 - \frac{1}{b}| \leq (\frac{1}{b})\frac{d}{2^{2p}}$. So it is sufficient if:

$$(\frac{1}{b})d/2^{2p} < (\frac{1}{b})/2^p - (\frac{1}{b})m_b/2^{2p}$$

Cancelling $(\frac{1}{b})/2^{2p}$ from both sides, we find that this is equivalent to:

$$d < 2^p - m_b$$

Consequently, the required relative error is guaranteed except possibly when $d \geq 2^p - m_b$, or equivalently $m_b \geq 2^p - d$, as claimed.

The HOL statement is as follows. Note that it uses $e = d/2^{2p}$ as compared with the statement we gave above, but this is inconsequential.

```
⊢  2 <= precision fmt ∧
   b ∈ iformat fmt ∧
   y ∈ iformat fmt ∧
   ¬(b = &0) ∧
   normalizes fmt b ∧
   normalizes fmt (inv(b)) ∧
   y = round fmt Nearest y0 ∧
   abs(y0 - inv(b)) <= e * abs(inv(b)) ∧
   e <= inv(&2 pow (precision fmt + 2)) ∧
   &(decode_fraction fmt b) <
   &2 pow (precision fmt) - &2 pow (2 * precision fmt) * e
   ⟹ abs(inv(b) - y) < abs(inv(b)) / &2 pow (precision fmt)
```

Thanks to this stronger theorem, we were actually able to design more efficient algorithms than those based on Markstein's original theorems, a surprising and gratifying effect of our formal verification project. For a more elaborate number-theoretic analysis of a related algorithm, see [30]; the proof described there has also been formalized in HOL Light.

Flag settings We must ensure not only correct results in all rounding modes, but that the flags are set correctly. However, this essentially follows in general from the correctness of the result in all rounding modes (strictly, in the case of underflow, we need to verify this for a format with slightly larger exponent range). For the correct setting of the inexact flag, we need only prove the following HOL theorem:

```
⊢  ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⟹ ∀rc. round fmt rc x = x ⇔ round fmt rc y = y
```

The proof is simple: if `x` rounds to itself, then it must be representable. But by hypothesis, `y` rounds to the same thing, that is `x`, in *all rounding modes*. In particular the roundings up and down imply `x <= y` and `x >= y`, so `y = x`. The other way round is similar.

4.3 Square root

Similarly, the Intel® Itanium® architecture defers square roots to software, and we have verified a number of sequences for the operation [29]. The process of formal verification follows a methodology established by Cornea [13]. A general analytical proof covers the majority of cases, but a number of potential exceptions are isolated using number-theoretic techniques and dealt with using an explicit case analysis. Proofs of this nature, large parts of which involve intricate but routine error bounding and the exhaustive solution of diophantine equations, are very tedious and error-prone to do by hand. In practice, one would do better to use *some* kind of machine assistance, such as *ad hoc* programs to solve the diophantine equations and check the special cases so derived. Although this can be helpful, it can also create new dangers of incorrectly implemented helper programs and transcription errors when passing results between 'hand' and 'machine' portions of the proof. By contrast, we perform all steps of the proof in HOL Light, and can be quite confident that no errors have been introduced.

In general terms, square root algorithms follow the same pattern as division algorithms: an initial approximation (now obtained by `frsqrta`) is successively refined, and at the end some more subtle steps are undertaken to ensure correct rounding. The initial refinement is similar in kind to those for reciprocals and quotients, though slightly more complicated, and we will not describe them in detail; interested readers can refer to [37, 12, 29]. Instead we focus on ensuring perfect rounding at the end. Note that whatever the final `fma` operation may be, say

$$S := S_3 + e_3 H_3$$

we can regard it, because of the basic IEEE correctness of the `fma`, as the rounding of the *exact* mathematical result $S_3 + e_3 H_3$, which we abbreviate S^*. Thanks to the special properties of the `fma`, we can design the initial refinement to ensure that the relative error in S^* is only a little more than 2^{-2p} where p is the floating-point precision. How can we infer from such a relative error bound that S will always be correctly rounded? We will focus on the round-to-nearest mode here; the proof for the other rounding modes are similar but need special consideration of earlier steps in the algorithm to ensure correct results when the result is exact ($\sqrt{0.25} = 0.5$ etc.)

Exclusion zones On general grounds we note that $\sqrt{a}$ cannot be exactly the mid-point between two floating-point numbers. This is not hard to see, since the square root of a number in a given format cannot denormalize in that format, and a non-denormal midpoint has $p+1$ significant digits, so its square must have more than p.[16]

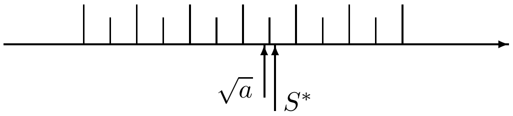
```
|- &0 <= a ∧ a ∈ iformat fmt ∧ b ∈ midpoints fmt
     ⟹ ¬(sqrt a = b)
```

This is a useful observation. We'll never be in the tricky case where there are two equally close floating-point numbers (resolved by the 'round to even' rule.) So in round-to-nearest, S^* and $\sqrt{a}$ could only round in different ways if there were a midpoint between them, for only then could the closest floating-point numbers to them differ. For example in the following diagram where large lines indicate floating-point numbers and smaller ones represent midpoints, $\sqrt{a}$ would round 'down' while S^* would round 'up':[17]

---

[16] An analogous result holds for quotients but here the denormal case must be dealt with specially. For example $2^{E_{min}} \times 0.111\cdots111/2$ is exactly a midpoint.

[17] Similarly, in the other rounding modes, misrounding could only occur if $\sqrt{a}$ and S^* are separated by a floating-point number. However we also need to consider the case when $\sqrt{a}$ *is* a floating-point number.



Although analyzing this condition combinatorially would be complicated, there is a much simpler sufficient condition. One can easily see that it would suffice to show that for any midpoint m:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* couldn't lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m ∈ midpoints fmt ⟹ abs(x - y) < abs(x - m))
   ⟹ round fmt Nearest x = round fmt Nearest y
```

One can arrive at an 'exclusion zone' theorem giving the minimum possible $|\sqrt{a} - m|$. However, this can be quite small, about $2^{-(2p+3)}$ relative to $\sqrt{a}$, where p is the precision. For example, when $p = 64$, consider the square root of the next floating-point number below 1, whose mantissa consists entirely of 1s. Its square root is about 2^{-131} from a midpoint:

$$\sqrt{1 - 2^{-64}} \approx (1 - 2^{65}) - 2^{-131}$$

Therefore, our relative error in S^* of rather more than 2^{-2p} is not adequate to justify perfect rounding based on the simple 'exclusion zone' theorem. However, our relative error bounds are far from sharp, and it seems quite plausible that the algorithm does nevertheless work correctly. What can we do?

One solution is to use more refined theorems [37], but this is complicated and may still fail to justify several algorithms that are intuitively believed to work correctly. An ingenious alternative developed by Cornea [13] is to observe that there are relatively few cases like $0.111 \cdots 1111$ whose square roots come close enough to render the exclusion zone theorem inapplicable, and these can be isolated by fairly straightforward number-theoretic methods. We can therefore:

- Isolate the special cases $a_1, \ldots, a_n$ that have square roots within the critical distance of a midpoint.
- Conclude from the simple exclusion zone theorem that the algorithm will give correct results except possibly for $a_1, \ldots, a_n$.
- Explicitly show that the algorithm is correct for the $a_1, \ldots, a_n$, (effectively by running it on those inputs).

This two-part approach is perhaps a little unusual, but not unknown even in pure mathematics.[18] For example, consider "Bertrand's Conjecture" (first proved by Chebyshev), stating that for any positive integer n there is a prime p

---

[18] A more extreme case is the 4-color theorem, whose proof relies on extensive (computer-assisted) checking of special cases [3].

with $n \leq p \leq 2n$. The most popular proof, originally due to Erdös [18], involves assuming $n > 4000$ for the main proof and separately checking the assertion for $n \leq 4000$.[19]

By some straightforward mathematics [13] formalized in HOL without difficulty, one can show that the difficult cases for square roots have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for some integer $|d| \leq D$, where D is roughly the factor by which the guaranteed relative error is excessive:

$$2^{p+2}m = k^2 + d \qquad 2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen $|d| \leq D$. For example, we might be interested in whether $2^{p+1}m = k^2 - 7$ has a solution. If so, the possible value(s) of m are added to the set of difficult cases. It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

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

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called 'Hensel lifting'). For example, if

$$2^{25}m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and deduce:

$$2^{24}m = 2k'^2 + 2k' - 3$$

By more even/odd reasoning, this has no solutions. In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable. One equation can split into two, but never more. For example, we have a formally proved HOL theorem asserting that for any double-extended number a,[20] rounding $\sqrt{a}$ and $\sqrt{a}(1+\epsilon)$ to double-extended precision using any of the four IEEE rounding modes will give the same results provided $|\epsilon| < 31/2^{131}$, with the possible exceptions of $2^{2e}m$ for:

$$\begin{aligned} m \in \{\, & 10074057467468575321, 10376293541461622781, \\ & 10376293541461622787, 11307741603771905196, \\ & 13812780109330227882, 14928119304823191698, \\ & 16640932189858196938, 18446744073709551611, \\ & 18446744073709551612, 18446744073709551613, \\ & 18446744073709551614, 18446744073709551615\} \end{aligned}$$

---

[19] An 'optimized' way of checking, referred to by [2] as "Landau's trick", is to verify that 3, 5, 7, 13, 23, 43, 83, 163, 317, 631, 1259, 2503 and 4001 are all prime and each is less than twice its predecessor.

[20] Note that there is more subtlety required when using such a result in a mixed-precision environment. For example, to obtain a single-precision result for a double-precision input, an algorithm that suffices for single-precision inputs may not be adequate even though the final precision is the same.

and $2^{2e+1}m$ for

$$m \in \{9223372036854775809, 9223372036854775811,\\ 11168682418930654643\}$$

Note that while some of these numbers are obvious special cases like $2^{64} - 1$, the "pattern" in others is only apparent from the kind of mathematical analysis we have undertaken here. They aren't likely to be exercised by random testing, or testing of plausible special cases.[21]

Checking formally that the algorithm works on the special cases can also be automated, by applying theorems on the uniqueness of rounding to the concrete numbers computed. (For a formal proof, it is not sufficient to separately test the implemented algorithm, since such a result has no formal status.) In order to avoid trying all possible even or odd exponents for the various significands, we exploit some results on invariance of the rounding and arithmetic involved in the algorithm under systematic scaling by 2^{2k}, doing a simple form of symbolic simulation by formal proof.

4.4 Transcendental functions

We have also proven rigorous error bounds for implementations of several common transcendental functions. Note that, according to current standard practice, the algorithms do not aim at perfect rounding, but allow a small additional relative error. Although ensuring perfect rounding for transcendental functions is possible, and may become standard in the future, it is highly non-trivial and involves at least some efficiency penalty [42]. We will consider here a floating-point *sin* and *cos* function; as will become clear shortly the internal structure is largely identical in the two cases.

As is quite typical for modern transcendental function implementations [56], the algorithm can be considered as three phases:

- Initial range reduction
- Core computation
- Reconstruction

For our trigonometric functions, the initial argument x is reduced modulo $\pi/2$. Mathematically, for any real x we can always write:

$$x = N(\pi/2) + r$$

where N is an integer (the closest to $x \cdot \frac{2}{\pi}$) and $|r| \leq \pi/4$. The core approximation is then a polynomial approximation to $sin(r)$ or $cos(r)$ as appropriate, similar to a truncation of the familiar Taylor series:

---

[21] On the other hand, we can well consider the mathematical analysis as a *source* of good test cases.

$$
\begin{aligned}
sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \\
cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots
\end{aligned}
$$

but with the pre-stored coefficients computed numerically to minimize the maximum error over r's range, using the so-called *Remez algorithm* [49]. Finally, the reconstruction phase: to obtain either $sin(x)$ and/or $cos(x)$, just return one of $sin(r)$, $cos(r)$, $-sin(r)$ or $-cos(r)$ depending on N modulo 4. For example:

$$sin((4M+3)(\pi/2)+r) = -cos(r)$$

Verification of range reduction The principal difficulty of implementing trigonometric range reduction is that the input argument x may be large and yet the reduced argument r very small, because x is unusually close to a multiple of $\pi/2$. In such cases, the computation of r needs to be performed very carefully. Assuming we have calculated N, we need to evaluate:

$$r = x - N\frac{\pi}{2}$$

However, $\frac{\pi}{2}$ is irrational and so cannot be represented exactly by any finite sum of floating point numbers. So however the above is computed, it must in fact calculate

$$r' = x - NP$$

for some approximation $P = \frac{\pi}{2} + \epsilon$. The relative error $\frac{|r'-r|}{|r|}$ is then $\frac{N|\epsilon|}{|r|}$ which is of the order $|\frac{x\epsilon}{r}|$. Therefore, to keep this relative error within acceptable bounds (say 2^{-70}) the accuracy required in the approximation P depends on how small the (true) reduced argument can be relative to the input argument. In order to formally verify the accuracy of the algorithm, we need to answer the purely mathematical question: how close can a double-extended precision floating point number be to an integer multiple of $\frac{\pi}{2}$? Having done that, we can proceed with the verification of the actual computation of the reduced argument in floating point arithmetic. This requires a certain amount of elementary number theory, analyzing *convergents* [4].

The result is that for nonzero inputs the reduced argument has magnitude at least around 2^{-69}. Assuming the input has size $\leq 2^{63}$, this means that an error of ϵ in the approximation of $\pi/2$ can constitute approximately a $2^{132}\epsilon$ relative error in r. Consequently, to keep the relative error down to about 2^{-70} we need $|\epsilon| < 2^{-202}$. Since a floating-point number has only 64 bits of precision, it would seem that we would need to approximate $\pi/2$ by four floating-point numbers $P_1, \ldots, P_4$ and face considerable complications in keeping down the rounding error in computing $x - N(P_1 + P_2 + P_3 + P_4)$. However, using an ingenious technique called *pre-reduction* [55], the difficulties can be reduced.

Verification of the core computation The core computation is simply a polynomial in the reduced argument; the most general *sin* polynomial used is of the form:[22]

$$p(r) = r + P_1 r^3 + P_2 r^5 + \cdots + P_8 r^{17}$$

where the P_i are all floating point numbers. Note that the P_i are not the same as the coefficients of the familiar Taylor series (which in any case are not exactly representable as floating point numbers), but arrived at using the Remez algorithm to minimize the worst-case error over the possible reduced argument range. The overall error in this phase consists of the approximation error $p(r) - sin(r)$ as well as the rounding errors for the particular evaluation strategy for the polynomial. All of these require some work to verify formally; for example we have implemented an automatic HOL derived rule to provably bound the error in approximating a mathematical function by a polynomial over a given interval. The final general correctness theorems we derive have the following form:

```
|- x ∈ floats Extended ∧ abs(Val x) <= &2 pow 64
   ⟹ prac (Extended,rc,fz) (fcos rc fz x) (cos(Val x))
          (#0.07341 * ulp(rformat Extended) (cos(Val x)))
```

The function `prac` means ‘pre-rounding accuracy’. The theorem states that provided x is a floating point number in the double-extended format, with $|x| \leq 2^{64}$ (a range somewhat wider than needed), the result excluding the final rounding is at most 0.07341 units in the last place from the true answer of $cos(x)$. This theorem is generic over all rounding modes `rc` and flush-to-zero settings `fz`. An easy corollary of this is that in round-to-nearest mode without flush-to-zero set the maximum error is 0.57341 ulps, since rounding to nearest can contribute at most 0.5 ulps. In other rounding modes, a more careful analysis is required, paying careful attention to the formal definition of a ‘unit in the last place’. The problem is that the true answer and the computed answer before the final rounding may in general lie on opposite sides of a (negative, since $|cos(x)| \leq 1$) power of 2. At this point, the gap between adjacent floating point numbers is different depending on whether one is considering the exact or computed result. In the case of round-to-nearest, however, this does not matter since the result will always round to the straddled power of 2, bringing it even closer to the exact answer.

5 Conclusion and future perspectives

Formal verification in this area is a good target for theorem proving. The work outlined here has contributed in several ways: bugs have been found, potential optimizations have been uncovered, and the general level of confidence and intellectual grasp has been raised. In particular, two key strengths of HOL Light are

---

[22] In fact, the reduced argument needs to be represented as two floating-point numbers, so there is an additional correction term that we ignore in this presentation.

important: (i) available library of formalized real analysis, and (ii) programmability of special-purpose inference rules without compromising soundness. Subsequent improvements might focus on integrating the verification more tightly into the design flow as in [44].

References

1. M. Aagaard and J. Harrison, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
2. M. Aigner and G. M. Ziegler. *Proofs from The Book*. Springer-Verlag, 2nd edition, 2001.
3. K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82:711–712, 1976.
4. A. Baker. *A Concise Introduction to the Theory of Numbers*. Cambridge University Press, 1985.
5. S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, ENS Lyon, 2004. Available on the Web from `http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf`.
6. W. S. Brown. A simple but realistic model of floating-point computation. *ACM Transactions on Mathematical Software*, 7:445–480, 1981.
7. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
9. A. Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
10. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.
11. C. W. Clenshaw and F. W. J. Olver. Beyond floating point. *Journal of the ACM*, 31:319–328, 1984.
12. M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing for Itanium Based Systems*. Intel Press, 2002.
13. M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.
14. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
15. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
16. T. J. Dekker. A floating-point technique for extending the available precision. *Numerical Mathematics*, 18:224–242, 1971.
17. C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 64(7):24–32, July 1998.

18. P. Erdös. Beweis eines Satzes von Tschebyshev. *Acta Scientiarum Mathematicarum (Szeged)*, 5:194–198, 1930.
19. E. Goldberg and Y. Novikov. BerkMin: a fast and robust Sat-solver. In C. D. Kloos and J. D. Franca, editors, *Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, pages 142–149, Paris, France, 2002. IEEE Computer Society Press.
20. M. J. C. Gordon. Representing a logic in the LCF metalanguage. In D. Néel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.
21. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
22. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
23. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
24. J. Harrison. Proof style. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer-Verlag.
25. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
26. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.
27. J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.
28. J. Harrison. Formal verification of IA-64 division algorithms. In Aagaard and Harrison [1], pages 234–251.
29. J. Harrison. Formal verification of square root algorithms. *Formal Methods in System Design*, 22:143–153, 2003.
30. J. Harrison. Isolating critical cases for reciprocals using integer factorization. In J.-C. Bajard and M. Schulte, editors, *Proceedings, 16th IEEE Symposium on Computer Arithmetic*, pages 148–157, Santiago de Compostela, Spain, 2003. IEEE Computer Society. Currently available from symposium Web site at `http://www.dec.usc.es/arith16/papers/paper-150.pdf`.
31. J. E. Holm. *Floating-Point Arithmetic and Program Correctness Proofs*. PhD thesis, Cornell University, 1980.
32. IEEE. 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, 1985.
33. C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, University of the Saarland, 2002. Available on the Web as `http://engr.smu.edu/~seidel/research/diss-jacobi.ps.gz`.
34. R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. In Aagaard and Harrison [1], pages 338–355.

35. S. Linnainmaa. Analysis of some known methods of improving the accuracy of floating-point sums. *BIT*, 14:167–202, 1974.
36. D. W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15:236–251, 1968.
37. P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, 2000.
38. P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.
39. O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.
40. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.
41. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.
42. J.-M. Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 1997.
43. J.-M. Muller. On the definition of $ulp(x)$. Research Report 2005-09, ENS Lyon, 2005.
44. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.
45. V. R. Pratt. Anatomy of the Pentium bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *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*, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.
46. D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California, Berkeley, 1992. Available on the Web as `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.
47. J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.
48. K. Quinn. Ever had problems rounding off figures? The stock exchange has. *Wall Street Journal*, November 8:?, 1983.
49. M. E. Remes. Sur le calcul effectif des polynomes d'approximation de Tchebichef. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 199:337–340, 1934.
50. D. Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
51. J. Sawada. Formal verification of divide and square root algorithms using series calculation. In D. Borrione, M. Kaufmann, and J. Moore, editors, *3rd International Workshop on the ACL2 Theorem Prover and its Applications*, pages 31–49. University of Grenoble, 2002.

52. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.
53. G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.
54. P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.
55. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In I. Koren and P. Kornerup, editors, *Proceedings, 14th IEEE symposium on on computer arithmetic*, pages 4–11, Adelaide, Australia, 1999. IEEE Computer Society.
56. P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the* 10^{th} *Symposium on Computer Arithemtic*, pages 232–236, 1991.
57. P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.
58. A. v. Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:68–81, 1966.
59. J. H. Wilkinson. *Rounding Errors in Algebraic Processes*, volume 32 of *National Physical Laboratory Notes on Applied Science*. Her Majesty's Stationery Office (HMSO), London, 1963.
60. N. Wirth. *Systematic Programming: An Introduction*. Prentice-Hall, 1973.