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

June 1997

Abstract

In that 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.

Our main theorem connects the floating point exponential to its abstract mathematical counterpart. The specification we prove is that the function has the correct overflow behaviour and, in the absence of overflow, the error in the result is less than 0.54 units in the last place (0.77 if the answer is denormalized) compared against the exact mathematical exponential function. The algorithm is expressed in a simple formalized programming language, intended to be a subset of real programming and hardware description languages. It uses underlying floating point operations (addition, multiplication etc.) that are assumed to conform to the IEEE-754 standard for binary floating point arithmetic.

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. We first present the main ideas and conclusions, and then collect some technical details about the prover and the underlying mathematical theories in appendices.

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

Contents

1 Introduction **1**

2 HOL Light **2**

3 Correctness criteria for floating point operations **4**
3.1 Round to nearest 5
3.2 Bounded error 6
3.3 Error commensurate with likely input error 7
3.4 Probabilistic and mixed criteria 8

4 Our implementation language **8**

5 Formalizing IEEE arithmetic **10**
5.1 Format parameters 10
5.2 Floating point numbers 11
5.3 Representation and valuation 12
5.4 Rounding 12
5.5 Arithmetic operations 14
5.6 Comparisons 15
5.7 Float and double types 16

6 Lemmas about floating point numbers **17**

7 The algorithm **19**

8 The HOL verification **21**
8.1 Checking of prestored constants 21
8.2 Error in range reduction 22
8.3 Error in polynomial approximation 22
8.4 Rounding errors 23
8.5 Overflow 24
8.6 Underflow 25

9 Conclusions and methodological remarks **25**

A HOL's deductive system **27**

B Syntax and semantics of our programming language **31**
B.1 Syntax of commands 31
B.2 Semantics of commands 32
B.3 Weakest preconditions 33
B.4 Verification condition generation 34

C Error in polynomial approximations **36**
C.1 Our approach 37
C.2 Polynomials in HOL 38
C.3 Squarefree decomposition 42
C.4 Sturm's theorem 43
C.5 Our application 45

D List of definitions and theorems **48**

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 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

[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.

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.

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:

[2] Pollack (1996) 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.

```
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 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,

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

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
$\bot$	`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 $P(x)$'. A quick summary of HOL's logical basics is given in an appendix; 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
`&`	(*none*)	Natural map $\mathbb{N} \to \mathbb{R}$
`--x`	$-x$	Unary negation of x
`inv(x)`	x^{-1}	Multiplicative inverse of x
`abs(x)`	$\lvert x \rvert$	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 purpose 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

[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.

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':

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

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 $0 \leq v(a) \leq v(a')$ and $0 \leq v(b) \leq v(b')$ then we also have $v(\mathrm{MUL}(a,b)) \leq v(\mathrm{MUL}(a',b'))$.

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 perhaps even analyzed and 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

[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.

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 $\leq 0.5\ ulp$.

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, 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.

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'. There is already plenty of experience of embedding such languages in HOL (Gordon 1989; von Wright, Hekanaho, Luostarinen, and Långbacka 1993), and our work follows the same general lines. 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.

For the detailed semantics of our language, we refer the reader to the appropriate appendix. However 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 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.[6] 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:[7]

```
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.

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.

[6] 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.

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

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 $emax = 2^{expwidth} - 1$.[8]
- 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
```

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.

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

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 $-\infty < finite < \infty$.[9] 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)
```

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.)

[9] 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.

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.[10]

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

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
```

[10] 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.

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 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:[11]

```
|- 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. 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 $Val : float \to \mathbb{R}$ (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:

[11] Note that $abs(-0) = -0$; the standard offers no guidance as to whether one should make $abs(-0) = +0$, but there is certainly an obvious case for it. This decision does not affect the algorithm considered here.

```
|- 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 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. 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 an appendix, 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.[12] 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

[12] 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.[13] 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.

[13] 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 is not specified in the 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.

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.

Theorems of this concrete nature are generally harder to prove than clean, abstract mathematical results. To give an ad hoc comparison, this proof 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.[14] 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

[14] 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`.

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.) 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.

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

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

Our insistence on reducing everything to logical first principles gives us the highest confidence in the final result, and has not been a disaster from the point of view of efficiency. There is one unfortunate exception: arithmetic. Performing arithmetic, particularly multiplication, of large numbers by proof is very time-consuming. So much so that the entire proof takes about 12 hours to run;[15] this is cut to about 2 hours if addition and multiplication are performed as primitives using CAML's

[15] 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.

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. 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).

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.

A HOL's deductive system

Here we will discuss the basics of the HOL types, logical syntax and deductive system. Gordon and Melham (1993) give a discussion that is both more leisurely and more precise, though the deductive system in HOL Light differs in detail from the one given there. HOL's logic is based on λ-calculus, a formalism invented by Alonzo Church. In HOL, as in λ-calculus, there are only four kinds of term:

- Constants, e.g. 1, 2, $\top$ (true) and $\bot$ (false). The set of constants is extensible by constant definition, which we discuss later.
- Variables, e.g. n, x, p. We assume some arbitrary denumerable set of variables; in HOL a variable may have any string as its name.
- Applications, e.g. $f(x)$. This represents the evaluation of a function f at an argument x; any terms may be used in place of f and x. The brackets are often omitted, e.g. $f\ x$.
- Abstractions, e.g. $\lambda x.\ t[x]$. This example represents 'the function of x that yields $t[x]$'. Any term may be used for the body; x need not appear free in it.

Abstractions are not often seen in informal mathematics, but they have at least two merits. First, they allow one to write down function-valued expressions without naming them (occasionally one sees $x \mapsto t[x]$ used for this purpose), and since our logic is avowedly higher order, it's desirable to place functions on an equal footing with first-order objects in this way. Secondly, they make variable dependencies and binding explicit; by contrast in informal mathematics one often writes $f(x)$ in situations where one really means $\lambda x.f(x)$. *Lambda-calculus* is, as its name suggests, a formal deductive system based on the above syntax, namely a system of rules for

generating equations $\vdash s = t$ (read the 'turnstile' symbol $\vdash$ as 'it is derivable that ...'). There are rules for the reflexivity and substitutivity of equality, as well as a few special lambda-calculus rules:

- Alpha-conversion means consistently changing the name of the variable in a λ-abstraction. It permits the deduction of equations like $\vdash (\lambda x.\, t[x]) = (\lambda y.\, t[y])$. On the intuitive semantics of lambda-expressions, this is obviously valid.
- Beta-conversion expresses the intention that abstraction and application are converse operations; it allows one to deduce equations like $\vdash (\lambda x.\, t[x])s = t[s]$.
- Eta-conversion incorporates a kind of extensionality principle, but expresses it purely in terms of the equation calculus. It allows the deduction of $\vdash (\lambda x.\, t\ x) = t$, where x is not free in t.

Church's original idea was to use λ-calculus as the core for a logical system, adding constants and additional rules for logical operations. For example, one can represent predication by function application, and truth by equality with the constant '$\top$'. Unfortunately this turned out to be inconsistent. For example, if N is a negation operation, one can derive the Russell paradox about the set of all sets that do not contain themselves (think of $P\ x$ as $x \in P$):[16]

$$\vdash (\lambda x.\, N(x\ x))(\lambda x.\, N(x\ x)) = N((\lambda x.\, N(x\ x))(\lambda x.\, N(x\ x)))$$

Accordingly, Church (1940) augmented λ-calculus with a theory of types, simplifying Russell's system from Principia Mathematica and giving what is often called 'simple type theory'. HOL follows this system quite closely. Every term has a unique *type* which is either one of the basic types or the result of applying a type constructor to other types. The only basic types in HOL are initially the type of booleans, *bool* and the infinite type of individuals *ind*; the only type operator is the function space constructor $\rightarrow$. HOL extends Church's system by allowing also 'type variables' which give a form of polymorphism. Examples of HOL types then, include *ind* and $\alpha \rightarrow bool$ (where α is a type variable). We write $t : \gamma$ to indicate that a term t has type γ. Readers familiar with set theory may like to think of types as sets within which the objects denoted by the terms live, so $t : \gamma$ can be read as $t \in \gamma$. Note that the use of the colon is already standard in set theory when used for function spaces, i.e. one typically writes $f : A \rightarrow B$ rather than $f \in A \rightarrow B$.

Just as with typed programming languages, functions may only be applied to arguments of the right type; only a function of type $f : \gamma \rightarrow \ldots$ may be applied to an argument of type γ. This restriction is decidable and applied at the level of the term syntax. In fact, when a term is written down, even with no type annotations at all, HOL is capable not only of deciding whether it has a type, but inferring a most general type for it if it does. Essentially the same form of type inference goes on in the ML programming language; both use an algorithm given by Milner (1978).[17]

HOL's logic is then built up by including constants for the usual logical operations. An attractive feature is that these do not need to be postulated, but can all be defined in terms of the underlying equation calculus — see Henkin (1963) for details. Actually HOL does take a few extra constants as primitive, for reasons that are partly historical and philosophical, partly practical. (For example, the HOL

[16] However this does motivate the fixpoint combinator $Y = \lambda P.\, (\lambda x.\, P(x\ x))(\lambda x.\, P(x\ x))$ which is of considerable interest in the theory of λ-calculus.

[17] ML also features so-called `let`-polymorphism which has no counterpart in HOL, and some versions also include equality types. Moreover, we have ignored operator overloading when asserting the most general type property.

definitions of the logical constants also work in the intuitionistic case once implication is taken as primitive.) The formal system allows the deduction of arbitrary *sequents* of the form $\phi_1, \ldots, \phi_n \vdash \psi$ (read as 'if ϕ_1 and ... and ϕ_n then ψ') where the terms involved have type *bool*; they need not simply be equations. However in principle all this could be implemented directly in the equation calculus. The additional primitive constants are implication, $\Longrightarrow$, which has type $bool \to bool \to bool$, and the Hilbert choice operator ε which has polymorphic type $(\alpha \to bool) \to \alpha$. The term $\varepsilon\ P$ denotes 'some x such that $P(x)$'. The primitive deductive system is based on the following rules. We assume the appropriate type restrictions without comment — for example the rule `MK_COMB` requires that the types of f and x match up as specified above, and if they do not it fails to apply.

$$\frac{}{\Gamma \vdash x = x}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash x = y}{\Gamma \vdash y = x}\ \texttt{SYM}$$

$$\frac{\Gamma \vdash x = y \quad \Delta \vdash y = z}{\Gamma \cup \Delta \vdash x = z}\ \texttt{TRANS}$$

$$\frac{}{\vdash (\lambda x.\ t[x])s = t[s]}\ \texttt{BETA_CONV}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x.\ t)}\ \texttt{ABS}$$

$$\frac{\Gamma \vdash f = g \quad \Delta \vdash x = y}{\Gamma \cup \Delta \vdash f(x) = g(y)}\ \texttt{MK_COMB}$$

$$\frac{}{p \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash q}{\Gamma - \{p\} \vdash p \Longrightarrow q}\ \texttt{DISCH}$$

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

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ_MP}$$

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

$$\frac{\Gamma \vdash p[x_1, \ldots, x_n]}{\Gamma \vdash p[t_1, \ldots, t_n]}\ \texttt{INST}$$

$$\frac{\Gamma \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma \vdash p[\gamma_1, \ldots, \gamma_n]}\ \texttt{INST_TYPE}$$

There are a few additional restrictions on these rules: in `ABS`, the variable x must not occur free in any of the assumptions Γ; likewise in `INST` and `INST_TYPE`

none of the term or type variables must occur free in Γ. Note that we should more properly write the conclusion of `ASSUME` as $\{p\} \vdash p$.

There are principles of definition for new constants and for new types and type operators. If t is a term with no free variables, and c is name not already in use as a constant symbol, one may add a new equational axiom of the form $\vdash c = t$.[18] Various more convenient methods of making definitions, e.g. inductive, primitive recursive or recursive over a wellfounded ordering, are packaged in the usual way as derived rules that ultimately make a simple definition of the above form.

Moreover, given any subset of a type γ, marked out by its characteristic predicate $P : \gamma \to bool$, then given a theorem asserting that P is nonempty, one can define a new type (or type operator if γ contains type variables) in bijection with this set. The type definition returns the appropriate type bijections. Once again, certain convenient kinds of type definitions such as the definition of enumerated, inductive and quotient types, are implemented automatically in terms of this primitive principle.

Both these definitional principles give a way of producing new mathematical theories without compromising soundness: one can easily prove that these principles are consistency-preserving. As an example, we shall show how the other logical constants are defined. These are $\top$ (true), $\forall$ (for all), $\exists$ (there exists), $\wedge$ (and), $\vee$ (or), $\bot$ (false) and $\neg$ (not). What we write as $\forall x.\, P[x]$ is a syntactic sugaring of $\forall(\lambda x.\, P[x])$. Using this technique, quantifiers and the Hilbert ε operator can be used as if they bound variables, but with all binding implemented in terms of λ-calculus.

$$
\begin{array}{rcl}
\top & = & (\lambda x.\, x) = (\lambda x.\, x) \\
\forall & = & \lambda P.\, P = \lambda x.\, \top \\
\exists & = & \lambda P.\, \forall Q.\, (\forall x.\, P(x) \Longrightarrow Q) \Longrightarrow Q \\
\wedge & = & \lambda p.\, \lambda q.\, \forall r.\, (p \Longrightarrow q \Longrightarrow r) \Longrightarrow r \\
\vee & = & \lambda p.\, \lambda q.\, \forall r.\, (p \Longrightarrow r) \Longrightarrow (q \Longrightarrow r) \Longrightarrow r \\
\bot & = & \forall P.\, P \\
\neg & = & \lambda t.\, t \Longrightarrow \bot
\end{array}
$$

These definitions look a bit obscure at first sight, but the first two express the intended meaning quite directly, while the next three correspond to the definitions of meets and joins in a lattice (think of implication as $\leq$). That concludes the logic proper, and in fact quite a bit of interesting mathematics, e.g. infinitary inductive definitions can be developed just from that basis, without, moreover, using the ε operator. But for general use we adopt three more axioms. First there is an axiom of extensionality, which we encode as an η-conversion theorem: $\vdash (\lambda x.\, t\ x) = t$. Then there is the axiom giving the basic property of the ε operator, that it picks out something satisfying P whenever there is something to pick:

$$\vdash \forall x.\, P(x) \Longrightarrow P(\varepsilon x.\, P[x])$$

This is a form of the Axiom of (global) Choice. Rather surprisingly, it also makes the logic classical, i.e. allows us to prove the theorem $\vdash \forall p.\, p \vee \neg p$; see Beeson (1984) for the proof we use. Finally there is an axiom asserting that the type *ind* is infinite. the Dedekind/Peirce definition of 'infinite' is used:

$$\vdash \exists f : ind \to ind.\, (\forall x_1, x_2.\, (f(x_1) = f(x_2)) \Longrightarrow (x_1 = x_2)) \wedge \neg(\forall y.\, \exists x.\, y = f(x))$$

[18] Subject to some restrictions on type variables which we will not enter into here. In fact HOL Light has a slightly different primitive constant definition rule, and this and others are derived from it. However little would change if things were implemented as above.

B Syntax and semantics of our programming language

In HOL jargon (Boulton et al. 1993) our programming language semantics is a *deep* embedding. This means that the (abstract) syntax of programs is represented in HOL and the denotation function is a formal HOL object. By contrast, in a shallow embedding, commands are interpreted directly as their semantics, with the denotation function purely a matter of 'syntactic sugar'. For our purposes, there is little to choose between the two, and a deep embedding sometimes allows us to prove general theorems because the class of commands is restricted.[19]

Our approach is quite similar to Nipkow (1996), but we do not concern ourselves directly with variables and storage locations. Instead, following von Wright, Hekanaho, Luostarinen, and Långbacka (1993) we work with an abstract state of type S. Human-readable expressions are implicitly abstracted over the state; in a sense, our embedding is shallow when it comes to expressions. For example, if the state consists of variables x, y and z, the expression $x + y$ is translated by the parser into $\lambda(x, y, z).\ x + y$, while the assignment $z := x + y$ is translated into a state mapping $\lambda(x, y, z).\ (x, y, x + y)$.

We will often be dealing with predicates on the state space, and it is convenient to 'lift' the standard logical operations up to this level to allow the most natural version of theorems. Accordingly, we make the following definitions, many of which correspond to standard set operations like intersection and complement. Logical tautologies at this 'lifted' level can be proved automatically using the simple HOL function `PRED_TAUT`.

```
|- False = (λx. F)

|- True = (λx. T)

|- ∀p. Not p = (λx. ¬p x)

|- ∀p q. p And q = (λx. p x ∧ q x)

|- ∀p q. p Or q = (λx. p x ∨ q x)

|- ∀P. Ands P = (λx. ∀p. P p ⟹ p x)

|- ∀P. Ors P = (λx. ∃p. P p ∧ p x)

|- ∀p q. p Imp q = (λx. p x ⟹ q x)
```

As well as this, we will use the following notion of inclusion of predicates, corresponding in fact to HOL's `SUBSET`:

```
|- ∀p q. p Implies q = (∀x. p x ⟹ q x)
```

B.1 Syntax of commands

The abstract syntax of commands is based on a recursive type with the following definition:

[19] A major defect of this approach in a typed system like HOL is that the state space is fixed throughout. Using a shallow embedding, it is much easier to treat blocks with local variables, procedures etc. See von Wright, Hekanaho, Luostarinen, and Långbacka (1993) for more details.

```
define_type
  "command = Assign (S->S)
           | Seq command command
           | Ite (S->bool) command command
           | While (S->bool) command";;
```

The constant `Assign` represents an assignment command. `Seq`, which has infix parsing status, is used to execute two commands in sequence; in the concrete syntax it is simply written as a semicolon. `Ite` stands for 'if ... then ... else ...', and accordingly takes an expression to test and two commands to choose between. Finally, `While` represents a while-loop. From the above basic commands, we create some others. These could be made primitive, but it simplifies some arguments if the number of primitive commands is small.

```
|- Skip = Assign I

|- Abort = While True Skip

|- If e c = Ite e c Skip

|- Do c e = c Seq (While e c)

|- Assert g = Ite g Skip Abort

|- Awhile i v e c = While e c

|- Ado i v c e = Do c e
```

These are, respectively, the 'skip' (do nothing) command, using the identity function `I`, the 'abort' command which disappears off into an infinite loop, a one-sided 'if ... then ...', a do-loop, where the condition is tested only at the end of the loop, and commands that are semantically identical to while and do loops but contain annotation information to indicate a loop invariant and variant.

B.2 Semantics of commands

The effect of a given command depends, in general, on the current state, i.e. the value of all the (relevant) variables at the time of execution. The outcome of the command is normally another state, but there is also the possibility that a program will loop indefinitely, or perhaps fail in some other way (e.g. owing to division by zero). We interpret a command semantically, following Gordon (1989), as the corresponding relation between initial and final states, a relation that is defined inductively via the following rules:

```
|- sem(Assign f) s (f s)

|- sem(c1) s s' ∧ sem(c2) s' s'' ⟹ sem(c1 Seq c2) s s''

|- e s ∧ sem(c1) s s' ⟹ sem(Ite e c1 c2) s s'

|- ¬(e s) ∧ sem(c2) s s' ⟹ sem(Ite e c1 c2) s s'

|- ¬(e s) ⟹ sem(While e c) s s

|- e s ∧ sem(c) s s' ∧ sem(While e c) s' s''
   ⟹ sem(While e c) s s''
```

This enumerates the different ways in which a command can cause a transition from one state to another. Because the definition is inductive, these are the *only* ways it can do so. In fact, we can prove that all our commands are deterministic, so from a given state, a command can cause a transition to at most one state:

```
|- sem(c) s s1 ∧ sem(c) s s2 ⟹ (s1 = s2)
```

A command may, however, fail to terminate, in which case there is no state into which a transition can occur. Our style of semantics extends easily to non-deterministic commands, but is unable to distinguish between possible and certain termination. For that, one needs a more sophisticated semantics, e.g. interpreting a command as a relation on $S \times S_\perp$ where $S_\perp$ is S augmented by an extra element to denote nontermination. See Hesselink (1992) for a relational semantics in that style, and Grundy (1993) for a discussion of the difficulties of our model in treating nondeterminism satisfactorily.

Although the definition of `sem` was highly operational, we can equally well take a denotational view and show how the semantics of a composite construct is built up from the semantics of its components. In fact, we prove such theorems as they are more convenient to work with later. For example, we have:

```
|- sem(Assign f) s s' = (s' = f s)

|- sem(If e c) s s' = e s ∧ sem c s s' ∨ ¬(e s) ∧ (s = s')

|- sem(While e c) s s' = sem(If e (c Seq While e c)) s s'
```

The last of these exhibits `While e c` as a fixpoint. We could prove in fact that it is the *least* fixpoint. However, for our purposes, proving the total correctness of particular programs, the mere knowledge that it is *a* fixpoint suffices. We will always establish that the state decreases according to some wellfounded 'variant' ordering round each iteration of a loop. Under these circumstances, it is easy to see that the fixpoint must be unique. This vindicates one's intuitive feeling that the above equation is enough for practical purposes. The knowledge that our correctness proofs for loops depend only on the above equation should make our semantics more plausible to the sceptic, since we believe it holds in far richer languages whose semantics can't be given in these simple terms.

B.3 Weakest preconditions

Now we can define the important notions of *weakest precondition* and *weakest liberal precondition* (Dijkstra 1976). Given a command c and a postcondition q (a predicate $S \to bool$ that we want to hold in the final state), the corresponding weakest precondition is the set of all states s from which the command c is guaranteed to terminate in a state satisfying q. The weakest *liberal* precondition is the set of states such that *if* the command terminates, then it does so in a state satisfying q. Formally:

```
|- terminates c s = ∃s'. sem c s s'

|- wlp c q s = ∀s'. sem c s s' ⟹ q s'

|- wp c q s = terminates c s ∧ wlp c q s
```

We define `wp` and `wlp` as curried functions, but otherwise our notation is similar to Dijkstra's `wp(c,q)` and `wlp(c,q)`. These functions are said to be *predicate*

transformers, since for each command they map a postcondition to a precondition, and each of these is a predicate.

There are certain properties of the *wp c* predicate transformers that follow easily from our definitions — the HOL proofs are all automatic after expanding a few definitions. These are given near the start of Dijkstra's book and have become known as healthiness conditions. The last one is, in Dijkstra's book, predicated on an assumption of determinism, but that is automatically the case here.

```
|- wp c False = False

|- q Implies r ==> wp c q Implies wp c r

|- wp c q And wp c r = wp c (q And r)

|- wp c p Or wp c q = wp c (p Or q))
```

In their work, von Wright, Hekanaho, Luostarinen, and Långbacka (1993) actually take the corresponding weakest precondition function *wp c* as the semantic definition of the command *c*. In some ways this is very elegant, with the predicate transformers being elements of a complete lattice, where the standard lattice operations have a natural reading in operational terms. Moreover, one can then consider different kinds of nondeterministic behaviour, 'demonic' and 'angelic', because the notion of satisfying a condition is made explicit. Very general unimplementable predicate transformers can be used in program development, which is especially convenient for stepwise refinement. However, we think it is, as a *definition*, less intuitive than the relational version that we adopt.

Nevertheless, weakest preconditions are the most convenient semantics to work with, so most of our early attention is devoted to deriving the weakest preconditions from the definitions of commands (rather than making them hold by definition). Once we have reached this stage, we never need to return to the fundamental semantics.

```
|- wp (Assign f) q = q o f

|- wp (c1 Seq c2) q = wp c1 (wp c2 q)

|- wp (Ite e c1 c2) q = (e And wp c1 q) Or (Not e And wp c2 q)

|- wp (If e (c Seq While e c)) q = wp (While e c) q

|- wp Skip q = q

|- wp Abort q = False

|- wp (If e c) q = (e And wp c q) Or (Not e And q)

|- wp (c Seq If e (Do c e)) q = wp (Do c e) q

|- wp (Assert g) q = g And q
```

Once again, the simple fixpoint characterization of loops is enough for us.

B.4 Verification condition generation

We can easily define the notions of total and partial correctness of programs using weakest preconditions:

```
|- correct p c q = p Implies (wp c q)

|- pcorrect p c q = p Implies (wlp c q)
```

Here, '`correct p c q`' means that 'if `c` is executed in a state satisfying `p`, then it will terminate in a state satisfying `q`'. This is known as *total correctness*, whereas the second notion, which we don't in fact use in the main body of work described here, represents *partial correctness*, i.e. 'if `c` is executed in a state satisfying `p`, then *if it terminates* it will do so in a state satisfying `q`'. Now we can prove the standard Floyd-Hoare rules for proving correctness assertions, e.g.

```
|- correct p c q' ∧ q' Implies q ⟹ correct p c q

|- (p Implies (λs. q(f s))) ⟹ correct p (Assign f) q

|- correct p c1 r ∧ correct r c2 q ⟹ correct p (c1 Seq c2) q

|- correct (p And e) c q ∧ (p And (Not e)) Implies q
   ⟹ correct p (If e c) q

|- WF($<<) ∧
   p Implies invariant ∧
   (Not e) And invariant Implies q ∧
   (∀X. correct
          (invariant And e And (λs. X = s)) c
          (invariant And (λs. s << X)))
   ⟹ correct p (While e c) q
```

For the while-loop, we require that the state decreases round each loop according to some wellfounded relation `<<`. Now to prove the correctness of a program, one can backchain through these theorems, supplying the appropriate values for intermediate assertions and variants. However, if the program is augmented with a few annotations, the machine can do all this automatically and leave us with only the verification conditions. This procedure is justified by versions of the above theorems.

Loops can only be dealt with automatically if a loop invariant and variant are both specified; this is the purpose of the special commands `Awhile` and `Ado`. Note that the special case `measure m` is the most common variant: this simply says that a natural-number-valued expression on the state decreases on each iteration. For example, we have:

```
|- WF($<<) ∧
   p Implies invariant ∧
   (Not e) And invariant Implies q ∧
   (∀X. correct
        (invariant And e And (λs. X = s)) c
        (invariant And (λs. s << X)))
   ⟹ correct p (Awhile invariant $<< e c) q

|- (e And invariant) Implies p ∧
   ((Not e) And invariant) Implies q ∧
   (∀X. correct
        (p And (λs. X = m(s))) c
        (invariant And (λs. m(s) < X)))
   ⟹ correct p (Ado invariant (measure m) c e) q
```

Now, in most cases, the correctness theorems can be applied automatically. For the case of a sequence of commands $c_1; \cdots; c_n$ (with an arbitrary order of association), some care is needed. First, if the user has supplied an assertion as one of the commands, then the sequence is reassociated[20] and the sequence split at that point, using the assertion as the intermediate condition:

```
|- p Implies b ∧ correct b c q ⟹ correct p (Assert b Seq c) q

|- correct p c b ∧ b Implies q ⟹ correct p (c Seq (Assert b)) q

|- correct p c b ∧ correct b c' q
   ⟹ correct p (c Seq (Assert b) Seq c') q`,
```

Otherwise, HOL tries to work out an appropriate intermediate property itself. It uses two approaches. First, there are special situations where the correct assertion *after* an assignment can be discovered from the precondition. Roughly speaking, it is sufficient that (a) the assignment does not affect the precondition, i.e. the precondition does not depend essentially on the value of the variable being assigned, and (b) that the expression being assigned to likewise does not involve the variable being assigned. Formally, these conditions are embedded in the following theorem. Note that condition (b) simply says that the assignment, considered as a state transition, is idempotent:

```
|- (p o f = p) ∧ (f o f = f) ∧ correct (p And (λs. s = f s)) c q
   ⟹ correct p (Assign f Seq c) q
```

In all other circumstances, HOL works backwards from the postcondition q, by trying to calculate $wp\ c_n\ q$, arriving at r, and breaking down the goal into two subgoals $correct\ p\ (c_1; \cdots; c_{n-1})\ r$ and $correct\ r\ c_n\ q$. Note that if r is really $wp\ c_n\ q$ the second goal is trivial. However HOL can't calculate it properly if c_n is a loop, so in that event, it just chooses the loop invariant.

In summary then, provided loops are properly annotated, VC generation is completely automatic. However the user can guide the form of the VCs by inserting additional assertions in the code.

C Error in polynomial approximations

Suppose we are interested in approximating a function $f(x)$ over a closed interval $[a, b]$, i.e. $\{x \mid a \leq x \leq b\}$, by a polynomial of degree n. The natural choice is the polynomial $p(x)$ that, out of all polynomials of degree n, has the best *minimax* error behaviour, i.e. minimizes the maximum magnitude of the error:

$$||f(x) - p(x)||_\infty = sup_{a \leq x \leq b} |f(x) - p(x)|$$

A fundamental theorem of Chebyshev states that such a polynomial always exists and has an interesting 'equal ripple' property: the maximum value of $|f(x) - p(x)|$ is attained at some $n + 2$ points in the interval such that the sign of $f(x) - p(x)$ alternates between successive points. However the theorem does not yield an analytical expression for the coefficients, or the maximum error. Certainly the truncated Taylor expansion is normally not the best approximation in this sense. The truncated expansion in terms of *Chebyshev polynomials* is usually close. In fact there are special cases, typically for rational functions, where the coefficients

[20] The semantics of commands is clearly independent of the order of association; this simply requires rewriting with the theorem about the weakest precondition of a sequence.

in such a series are those for a best approximation, or can be converted into them analytically — see Rivlin (1962) for example. But these are no good for the functions we are interested in, and the only general methods known are iterative numerical ones that rely on numerically approximating extrema as part of the intermediate steps. The standard algorithm is usually attributed to Remes (1934), and also often referred to as the *exchange algorithm*.

In summary, then, it seems difficult to calculate the maximum error in the best approximation by analytical methods. Besides, any value so found would need slight but messy modifications when the coefficients are stored as approximate floating point numbers; typically the required coefficients will not be exactly representable. Instead, we allow the approximation to be calculated outside the theorem prover by arbitrary means, with coefficients truncated to the form actually used. Then we prove a theorem characterizing the maximum error in this polynomial approximation. Our procedure has the added advantage that we can take from the literature the polynomial approximations actually considered by other workers, without worrying about whether they coincide with the best approximations as we would calculate them.

C.1 Our approach

The function $f(x)$ that we are interested in approximating is typically fairly well-behaved, and therefore so is the error $e(x) = f(x) - p(x)$ for a polynomial $p(x)$. In particular, we can normally assume that $e(x)$ is differentiable at least once throughout the interval. Moreover, it is usually easy to arrive at a (pessimistic) bound B for the absolute value of the derivative $e'(x)$. This isn't the case for pathological functions such as $xsin(1/x)$ near zero, but for the basic transcendentals, it is easy over the intervals typically considered. By the mean value theorem (proved in HOL), this ensures that B is a fixed bound for uniform continuity, meaning that:

$$\forall x, x' \in [a, b].\ |e(x) - e(x')| \leq B|x - x'|$$

Therefore it is easy in principle to approximate the extremal values of $e(x)$ to an accuracy ϵ simply by dividing up the interval into a set of values a_i at most ϵ/B apart and finding the maximum of the $e(a_i)$. In fact, this method works for a weaker notion of uniform continuity, normally assumed in the definition of 'computable real function', and for functions of more than one variable (Pour-El and Richards 1980).

Nevertheless, the above is too expensive in practice, since it involves a large number of calculations. This is all the more true in HOL, where a single evaluation of a transcendental function to moderate accuracy can take many seconds. A much more refined approach is to use the fact that if a function is differentiable in an interval, then its maximum and minimum are each attained either at the endpoints of the interval or at one of the points of zero derivative. This is easy to prove in HOL using the existing real analysis theory. So if we can approximate the points of zero derivative to within ϵ/B, we can get the maximum and minimum to accuracy ϵ by evaluating $e(x)$ only at these points.

However, we are now faced with the problem of locating the points of zero derivative. It's generally pretty easy in practice to approximate all the derivative's roots x_i numerically, e.g. using a Newton iteration. Provided they are simple roots, we can then find (rational) numbers α_i and β_i such that $\alpha_i < x_i < \beta_i$ and the signs of $f(\alpha_i)$ and $f(\beta_i)$ differ. Now the intermediate value theorem (already proved in HOL) assures us that there is a root between α_i and β_i. If we also ensure that $\beta_i - \alpha_i \leq \epsilon/B$ this approximation serves for the later calculations. However there are two difficulties. First, if x_i is a double root, or in general a root of even order, we will no longer have different signs for $f(\alpha_i)$ and $f(\beta_i)$ and it will be more

complicated to prove in HOL that there is a root between those points. A more serious problem is that we have no HOL proof that we have located *all* the roots of $e'(x)$, and without that we have no way of proving our final result.

We are not aware of any simple general theory that can prove exactly how many roots an equation involving transcendental functions has. Therefore the above approach seems very difficult. Instead we modify it as follows.[21] We approximate $f(x)$ by a truncated Taylor series $t(x)$, choosing the degree so large that truncation incurs an error well below the tightness of the bound we are trying to achieve, say $\epsilon/2$. (An alternative would be to truncate the Chebyshev expansion, but although that might lead to a lower-degree polynomial, the analytical details are messier.) Now instead of working with $f(x)$ above we work with $t(x)$, and we then need only locate all the zeros of a *polynomial* $e'(x)$. Moreover this polynomial has rational coefficients, since floating point numbers are rational and the coefficients in all the typical Taylor series are rational. This problem is tractable. It is not hard to prove (see later) that a polynomial of degree n can have at most n roots. If we find exactly n, our task is necessarily complete. This favourable situation can't be relied on, but in general we can calculate the exact number of roots using a classical result due to Sturm. The main difficulties are that we must prove this theorem in HOL and then apply it.

C.2 Polynomials in HOL

The ring $\mathbb{R}[X]$ of univariate polynomials over the reals may be characterized abstractly via a universal property. The most natural concrete definition is that it is the space of functions $c : \mathbb{N} \to \mathbb{R}$ such that for all sufficiently large n we have $c_n = 0$.[22] We define the degree of c, written $\partial(c)$ or $deg(c)$, to be the least d such that $\forall i > d.\ c_i = 0$. The function c is thought of as $\Sigma_{i=0}^{deg(c)} c_i x^i$, and this motivates the definition of addition and negation componentwise, e.g. $(b + c)_i = b_i + c_i$, and multiplication by:

$$(bc)_n = \Sigma_{i=0}^{n} b_i c_{n-i}$$

Rather than define a type of polynomials in this way, we deal directly with the *functions* $\mathbb{R} \to \mathbb{R}$ determined by polynomials. For a polynomial c, this is the function that takes x to $\Sigma_{i=0}^{deg(c)} c_i x^i$. Actually, in the case of the reals, nothing is lost by this, since every polynomial function determines a unique polynomial, and vice versa.[23] We define a function `poly` that maps a list of coefficients $[c_0; \ldots; c_n]$ into the corresponding polynomial function $\lambda x.\ \Sigma_{i=0}^{n} c_i x^i$:

```
|- (poly [] x = &0) ∧ (poly (CONS h t) x = h + x * poly t x)
```

Note that `poly` is not injective, since lists with trailing zeros give the same function, and so the salient notion of equality for polynomial functions arising from two lists l_1 and l_2 is that `poly` l_1 = `poly` l_2, or equivalently $\forall x.$`poly` l_1 x = `poly` l_2 x, rather than simply $l_1 = l_2$. If desired, we could introduce an equivalence relation between lists corresponding to this notion. We can now define arithmetic operations on lists by primitive recursion, including addition (`++`):

```
|- ([] ++ l2 = l2) ∧
   (CONS h t ++ l2 = if (l2 = []) then CONS h t
                     else CONS (h + (HD l2) (t ++ (TL l2))))
```

[21] Thanks to David Wheeler for this suggestion.

[22] We write c_n rather than $c(n)$, since it is probably more natural.

[23] This depends on the fact that the underlying ring is infinite. For example in a 2-element ring the polynomials x and x^2 are distinct even though they both determine the same function.

multiplication by a constant (##):

```
|- (c ## [] = []) ∧
   (c ## CONS h t = CONS (c * h) (c ## t))
```

and multiplication of two polynomials (**):

```
|- ([] ** l2 = []) ∧
   (CONS h t ** l2 =
      if t = [] then h ## l2 else h ## l2 ++ CONS (&0) (t ** l2))
```

From these, we can define negation and exponentiation in an obvious way:

```
|- neg p = --(&1) ## p

|- (p exp 0 = [&1]) ∧
   (p exp (SUC n) = p ** p exp n)
```

We also need to define differentiation of polynomials. This can be expressed purely as a formal manipulation of the list of coefficients, conveniently done via an auxiliary function:

```
|- (diff_aux n [] = []) ∧
   (diff_aux n (CONS h t) = CONS (&n * h) (diff_aux (SUC n) t))

|- diff l = if l = [] then [] else diff_aux 1 (TL l)
```

We then prove that all these operations do indeed work as expected on the corresponding polynomial functions. These are all straightforward single or double structural inductions on lists:

```
|- ∀p1 p2 x. poly (p1 ++ p2) x = poly p1 x + poly p2 x

|- ∀p c x. poly (c ## p) x = c * poly p x

|- ∀p x. poly (neg p) x = --(poly p x)

|- ∀x p1 p2. poly (p1 ** p2) x = poly p1 x * poly p2 x

|- ∀p n x. poly (p exp n) x = (poly p x) pow n

|- ∀l x. ((poly l) diffl (poly (diff l) x)) x
```

The last of these uses a notion from the HOL real analysis theory: it says that `poly` l is locally differentiable at x with derivative `poly` (`diff` l) x there. From the above results, it is straightforward to verify all the basic properties that one would expect of polynomial functions, e.g. that addition and multiplication are commutative, and that all polynomial functions are infinitely continuously differentiable, and that the product rule holds for derivatives of polynomial products. Note a subtle point however: not all identities hold at the level of *lists*. For example we have:

```
|- ([] ** [&1; &2] = []) ∧
   ([&1; &2] ** [] = [&0])
```

We do however define a function that deletes trailing zeros, and prove that it does not affect the polynomial function. Proving the opposite, that if l_1 and l_2 give rise to the same polynomial function then their normalized forms are equal, is harder, and it's convenient to wait till we've proved the key property that a nontrivial polynomial only has finitely many roots.

```
|- (normalize [] = []) ∧
   (normalize (CONS h t) = if normalize t = [] then
                             if h = &0 then [] else [h]
                           else CONS h (normalize t))

|- ∀p. poly (normalize p) = poly p
```

If desired, we could redefine the addition operation to use this at the end, and then all operations would return normalized results for normalized inputs. However we do not do that, merely using it to define the notion of degree:

```
|- ∀p. degree p = PRE (LENGTH (normalize p))
```

Although we don't really want to rely on representation details, it's very convenient to get us started to prove one key result at the level of lists, rather than polynomial functions. This is that if a polynomial derived from a nonempty list is divided by a linear polynomial, there is a constant remainder:

```
|- ∀t h. ∃q r. CONS h t = [r] ++ [--a; &1] ** q
```

(Note that `[--a; &1]` is thought of as '$x - a$'.) From this, we find that if a is a root of a nontrivial polynomial, then the polynomial is divisible by $(x - a)$ *at the level of lists*:

```
|- ∀a p. (poly p a = &0) = (p = []) ∨ (∃q. p = [--a; &1] ** q)
```

Now since we also have:

```
|- ∀q. LENGTH ([--a; &1] ** q) = SUC (LENGTH q)
```

it is a straightforward induction on the length of a generating list to show that a nonzero polynomial can only have finitely many roots, bounded, in fact, by the length of the list, and therefore by the degree of a normalized polynomial:

```
|- ∀p. ¬(poly p = poly [])
       ⟹ (∃i. ∀x. (poly p x = &0)
                   ⟹ (∃n. n <= (LENGTH p) ∧ (x = i n)))
```

or more abstractly, using a notion from the HOL set theory:

```
|- ∀p. ¬(poly p = poly []) ⟹ FINITE {x | poly p x = &0}
```

Several key results flow immediately from this fact. The polynomial ring has no zerodivisors, i.e. if the product of two polynomials is trivial, so is one of the factors. Hence cancellation holds. We can also get a more constructive definition of what it means for a polynomial to be zero.

```
|- ∀p q. (poly (p ** q) = poly []) =
         (poly p = poly []) ∨ (poly q = poly [])

|- ∀p q r. (poly (p ** q) = poly (p ** r)) =
           (poly p = poly []) ∨ (poly q = poly r)

|- ∀p. (poly p = poly []) = FORALL (λc. c = &0) p
```

where here, and later, we use the logical operations on lists defined by:

```
|- (FORALL P [] = T) ∧
   (FORALL P (CONS h t) = P h ∧ FORALL P t)

|- (EX P [] = F) ∧
   (EX P (CONS h t) = P h ∨ EX P t)
```

After this, there are no real mathematical difficulties left in getting the results we want; it is merely necessary to accumulate a number of lemmas. Many of these are concerned with divisibility of polynomials, which we define, no longer at the list level, by:

```
|- p1 divides p2 = ∃q. poly p2 = poly (p1 ** q)
```

There are various obvious properties collected, e.g:

```
|- ∀p. p divides p

|- ∀p q r. p divides q ∧ q divides r ⟹ p divides r

|- ∀p q m n. p exp n divides q ∧ m <= n ⟹ p exp m divides q

|- ∀p q r. p divides q ∧ p divides r ⟹ p divides q ++ r

|- ∀p q r. p divides q ∧ p divides q ++ r ⟹ p divides r

|- ∀p q. (poly p = poly []) ⟹ q divides p
```

A slightly less trivial property is that the linear polynomials are prime elements in the polynomial ring:

```
|- ∀a p q. [a; &1] divides p ** q =
           [a; &1] divides p ∨ [a; &1] divides q
```

A major result in what follows is that for each nonzero polynomial p and real number a, we have a welldefined 'order' n for a such that $p(x)$ is divisible by $(x-a)^n$ but not by $(x-a)^{n+1}$. Hence a is a root of p precisely if its order is nonzero. The definition as a choice term is trivial:

```
|- order a p =
       εn. [--a; &1] exp n divides p ∧
           ¬([--a; &1] exp SUC n divides p)
```

Some tedious but straightforward proofs are required to show that it has the required properties for any nontrivial polynomial:

```
|- ∀p a. ¬(poly p = poly [])
         ⟹ ([--a; &1] exp (order a p)) divides p ∧
             ¬(([--a; &1] exp (SUC(order a p))) divides p)

|- ∀p a. (poly p a = &0) = (poly p = poly []) ∨ ¬(order a p = 0)

|- ∀p a n. [--a; &1] exp n divides p =
           (poly p = poly []) ∨ n <= order a p

|- ∀p a. ¬(poly p = poly [])
         ⟹ (∃q. (poly p = poly ([--a; &1] exp (order a p) ** q)) ∧
                 ¬([--a; &1] divides q))
```

C.3 Squarefree decomposition

Given a polynomial $p(x)$, we are interested in locating its roots. As we have already mentioned, given an isolating interval $[\alpha, \beta]$ for a root such that $p(\alpha)$ and $p(\beta)$ have opposite signs, we can straightforwardly prove in HOL from the intermediate value theorem and the known continuity of polynomials that there must be at least one root in the interval. However, this doesn't work for roots of even order (in the precise sense of 'order' we have defined above), since in that case the function has the same sign at either side of the root. This is one reason why we would prefer the polynomial to have no multiple real roots. Another reason is that Sturm's theorem is easier to prove for polynomials without multiple real roots — this is actually the only form we have proved in HOL.

Therefore, it's convenient to start off by finding the so-called 'squarefree decomposition' of p. To get this, we divide p by $gcd(p, p')$ where p' is the derivative polynomial. It is easy to see that the resulting polynomial has the same roots but that they are all of order 1. If a is a root of $p(x)$ of order $n+1$, then we have $p(x) = (x-a)^{n+1}q(x)$ for $q(a) \neq 0$. Differentiating, we get $p'(x) = (n+1)(x-a)^n q(x) + (x-a)^{n+1}q'(x)$. This is obviously divisible by $(x-a)^n$ but not by $(x-a)^{n+1}$, for that would, by the primality of linear factors, imply that $(x-a)$ divides $q(x)$, a contradiction since we assumed $q(a) \neq 0$. Hence a has order $min(n, n+1) = n$ in $gcd(p, p')$, and so order 1 in $p/gcd(p, p')$.

We start by defining what it means for a polynomial to be 'squarefree', i.e. to have no quadratic factors. Actually, since we only consider real factors, we prepend the letter 'r':

```
|- ∀p. rsquarefree p =
       ¬(poly p = poly []) ∧ (∀a. (order a p = 0) ∨ (order a p = 1))
```

We prove without much trouble that this is equivalent to the fact that p and p' have no common factors, and deduce a simple decomposition theorem for squarefree polynomials:

```
|- ∀p. rsquarefree p =
       (∀a. ¬((poly p a = &0) ∧ (poly (diff p) a = &0)))

|- ∀p a.
      rsquarefree p ∧ (poly p a = &0)
      ⟹ (∃q. (poly p = poly ([--a; &1] ** q)) ∧ ¬(poly q a = &0))
```

Now we need to show that dividing by $gcd(p, p')$ always converts a polynomial p into a squarefree one with the same roots. Rather than defining $gcd(p, p')$ explicitly in HOL, we will simply assume some d such that the following hold for some polynomials r and s, q and e:

$$\begin{aligned} p &= qd \\ p' &= ed \\ d &= rp + sp' \end{aligned}$$

It is clear that this constrains d to be a gcd of p and p', since any other common divisor of p and p' must divide d. We can calculate the gcd externally together with the coefficients r and s, using the `gcdex` function of the Maple computer algebra system, and get q and e via its division function. These are then plugged into the following HOL theorem for checking.

```
|- ∀p q d e r s.
      ¬(poly (diff p) = poly []) ∧
      (poly p = poly (q ** d)) ∧
      (poly (diff p) = poly (e ** d)) ∧
      (poly d = poly (r ** p ++ s ** diff p))
      ⟹ rsquarefree q ∧
          (∀a. (poly q a = &0) = (poly p a = &0))
```

The proof of this is not too difficult, by deriving a couple of additional facts about how orders of roots interact with operations:

```
|- ∀a p q.
      ¬(poly (p ** q) = poly [])
      ⟹ (order a (p ** q) = order a p + order a q)

|- ∀p a.
      ¬(poly (diff p) = poly []) ∧ ¬(order a p = 0)
      ⟹ (order a p = SUC(order a (diff p)))
```

Now fix a and consider the orders of a in the various polynomials we consider above. We have, assuming p and p' are nontrivial (the latter obviously implies the former):

$$\begin{aligned} order_a(p) &= order_a(q) + order_a(d) \\ order_a(p') &= order_a(e) + order_a(d) \\ order_a(d) &\geq min(order_a(p'), order_a(p)) \end{aligned}$$

These together imply (over the natural numbers) that $order_a(q) = 0$ if $order_a(p) = 0$ and $order_a(q) = 1$ otherwise. The proof is done automatically by HOL's linear natural number arithmetic package, after case-splitting over whether $order_a(p)$ is zero and throwing in the fact that $order_a(p) = order_a(p') + 1$ in the latter case.

C.4 Sturm's theorem

Sturm's theorem (Benedetti and Risler 1990) gives a precise figure for the number of (distinct) real roots a polynomial has in an interval. Assuming the polynomial has rational coefficients and the endpoints of the interval are rational, it requires only rational arithmetic. The key concept is a *Sturm sequence* for a polynomial p. This is a finite sequence of polynomials, with p as the first element that has certain important properties. Rather than deal abstractly in terms of these properties, we use the 'standard' Sturm sequence directly. This starts with $p_0 = p$ and $p_1 = p'$ and thereafter proceeds by division, with a change of sign, so that $p_i = q_i p_{i+1} - p_{i+2}$ for some quotient q_i, and $deg(p_{i+2}) < deg(p_{i+1})$. Actually we can relax this slightly by rescaling the polynomials to keep the coefficients as simple as possible, provided we do not change their signs. We thus define the property of being a Sturm sequence in HOL as follows.

```
|- (STURM p p' [] = p' divides p) ∧
   (STURM p p' (CONS g gs) = (∃k. &0 < k ∧ p' divides (p ++ k ## g)) ∧
                             degree g < degree p' ∧
                             STURM p' g gs)
```

Note that we separate out the first two elements of the list, to give simpler manipulation, and do not yet assume that p' is the derivative of p. We next define

the number of variations in sign of a finite sequence of numbers (or a list in our formalization), not counting zeros:

```
|- (varrec prev [] = 0) ∧
   (varrec prev (CONS h t) =
      if prev * h < &0 then SUC (varrec h t)
      else if h = &0 then varrec prev t else varrec h t)

|- variation l = varrec (&0) l
```

Sturm's theorem involves calculating the number of variations in sign of the polynomials in a Sturm sequence, evaluated at a point. It asserts that the number of roots of p between a and b is the difference in this variation when calculated at a and b respectively. The proof proceeds by analyzing how this variation changes across roots of polynomials in the Sturm sequence. We can break the problem down so that we only need to consider one root at a time. It is easy to prove by induction on the definition of finiteness that any finite set of reals can be laid out in an ordered linear sequence $i_0, \ldots, i_{N-1}$:

```
|- ∀s. FINITE s
       ⟹ (∃i N. (∀x. x IN s = (∃k. k < N ∧ (x = i k))) ∧
                 (∀k. i k < i (SUC k)))
```

The set of zeros of a list of nontrivial polynomials is finite (optionally in any interval, since any subset of a finite set is finite), by list induction, so we can find such an enumeration of the points that are roots of any of the polynomials in a sequence. Now by considering the mid-points between adjacent i_k, we can split up an interval $[a, b]$ into intervals so that there is at most one root in each (though it may be a root of more than one of the polynomials at once). This is actually quite tedious to prove, since we need to take care at the endpoints.

Now we can confine ourselves to intervals containing at most one root, and such that if the root is one of the endpoints, it is not a root of the starting polynomial p. (This is ruled out for the two endpoints a and b by hypothesis; for internal endpoints this holds for the other polynomials too.) We just need to prove that if such an interval contains no root of p, the variation is the same at both ends, while if there is a root, it decreases by 1 in passing from left to right.

The first is pretty easy, under fairly weak hypotheses. Consider an interval $[a, b]$ and suppose that $p_i(c) \neq 0$ but $p_{i+1}(c) = 0$. Since we have $p_i = q_i p_{i+1} - k p_{i+2}$ for positive k, this means that $p_i(c)$ and $p_{i+2}(c)$ have opposite signs. Since c is the only possible root, neither p_i nor p_{i+2} changes sign throughout the interval, so they have opposite signs at both ends. Now the signs of $p_{i+1}(a)$ and $p_{i+1}(b)$ make no difference to the overall variation between p_i and p_{i+2}, which is one in each case. This argument is easily formalized in HOL using induction, though not pure structural induction. Depending on whether the head element of a list is zero, we either move to the consideration of a list one element or two elements shorter, so we need to perform wellfounded induction on the length of the list. The final result is:

```
|- ∀l f f' c.
       STURM f f' l ∧ a <= c ∧ c <= b ∧
       (∀x. a <= x ∧ x <= b ∧
            EX (λp. poly p x = &0) (CONS f (CONS f' l))
            ⟹ (x = c)) ∧
       ¬(poly f c = &0)
       ⟹ (varrec (poly f a) (MAP (λp. poly p a) (CONS f' l)) =
            varrec (poly f b) (MAP (λp. poly p b) (CONS f' l)))
```

We just need to prove that if the starting polynomial does have a root in the interval, then the variation changes by 1. Now we use an assumption that the starting polynomial is squarefree. Therefore the derivative has no zero in the interval, and so by the above lemma we get no change in variation from the tail of the sequence. We need only consider the change in sign from $p(x)$ to $p'(x)$. In fact there must be exactly one change of sign, because p crosses the axis precisely once, the root being simple. This is easily formalized using the mean value theorem: the derivative does not change sign over the interval, so either the derivative is positive everywhere and the function goes from negative to positive, or vice versa. In either case the result follows. Plugging this together with the lemma for the rest of the sequence, we get the result for a single interval, and so by summing over the intervals, we get the final result:

```
|- ∀f a b l.
      a <= b ∧ ¬(poly f a = &0) ∧ ¬(poly f b = &0) ∧
      rsquarefree f ∧ STURM f (diff f) l
      ⟹ {x | a <= x ∧ x <= b ∧ (poly f x = &0)} HAS_SIZE
          (variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))) -
           variation (MAP (λp. poly p b) (CONS f (CONS (diff f) l))))
```

where, because the HOL cardinality function is total and hence does not encode finiteness, we use:

```
|- s HAS_SIZE n = FINITE s ∧ (CARD s = n)
```

The main result also holds without the restriction that the starting polynomial be squarefree, but it is more complicated to prove (using the above squarefree case as a lemma) and we do not need it. However, note that the Sturm sequence is up to rescaling a Euclidean division sequence for the polynomial and its derivative. Hence we expect the last term of the Sturm sequence to be a constant polynomial, and this itself implies that the original polynomial must have been squarefree without a separate check. Thus we can sharpen the above slightly, if we rule out a few degenerate cases. The form we actually use, as proved in HOL, is:

```
|- ∀f a b l d.
       a <= b ∧
       ¬(poly f a = &0) ∧
       ¬(poly f b = &0) ∧
       ¬(poly (diff f) = poly []) ∧
       STURM f (diff f) l ∧
       ¬(l = []) ∧
       (LAST l = [d]) ∧
       ¬(d = &0)
       ⟹ {x | a <= x ∧ x <= b ∧ (poly f x = &0)} HAS_SIZE
           variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))) -
           variation (MAP (λp. poly p b) (CONS f (CONS (diff f) l)))
```

C.5 Our application

Tang's algorithm uses polynomials to approximate $e^x - 1$ over a range slightly larger than $[-\frac{ln(2)}{64}, \frac{ln(2)}{64}]$; as already mentioned, we use $[-0.010844, +0.010844]$ for single precision. Tang's paper gives the coefficients as IEEE numbers in hexadecimal form. Translated into rational numbers, the approximating polynomial for single precision is:

$$p_{single}(x) = x + \frac{8388676}{2^{24}}x^2 + \frac{11184876}{2^{26}}x^3$$

Tang asserts an error of approximately $2^{-33.2}$ for the approximation, obtained by 'locating numerically all the extreme points of $e^t - 1 - p(t)$'. It is this result that we will formalize in HOL. Consider the error in truncated Taylor expansions. There are several versions of Taylor's theorem proved in the HOL real analysis theory, and the most convenient one for our purposes is the one for infinitely everywhere differentiable functions:

```
|- ∀f diff.
       (diff 0 = f) ∧ (∀m x. (diff m diffl diff (SUC m) x) x)
       ⟹ (∀x n.
             ∃t. abs t <= abs x ∧
                 (f x =
                  Sum (0,n) (λm. diff m (&0) / &(FACT m) * x pow m) +
                  diff n t / &(FACT n) * x pow n))
```

Note that `Sum (0,n) f` means $\Sigma_{i=0}^{n-1} f_i$, so the above says that for some t with $|t| \leq |x|$ we have

$$f(x) = (\Sigma_{i=0}^{n-1} \frac{f^{(i)}(0)}{i!}x^i) + \frac{f^{(n)}(t)}{n!}x^n$$

We can sharpen the above to $|t| < |x|$ when $x \neq 0$ and $n \neq 0$, but that is unnecessary here. Instantiating to the exponential function and using the known derivative, we get:

```
|- ∀x n. ∃t. abs t <= abs x ∧
            (exp x = Sum (0,n) (λm. x pow m / &(FACT m)) +
                     exp t / &(FACT n) * x pow n)
```

that is, $e^x = (\Sigma_{i=0}^{n-1} x^i/i!) + e^t x^n/n!$. We need only consider x with $|x| \leq ln(2)/64$ (approximately). In this case, e^t for $|t| \leq |x|$ is almost 1, so the maximum error from truncating the series before the nth term is approximately $\epsilon_n = (ln(2)/64)^n/n!$. We can see that an additional 3 terms gives an error of about $\epsilon_7 = 2^{-58}$, easily small enough for our purposes. [24]

$$t(x) = x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5 + \frac{1}{720}x^6$$

Hence the error polynomial $e(x) = t(x) - p_{single}(x)$ is:

$$e(x) = -\frac{17}{4194304}x^2 - \frac{49}{50331648}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5 + \frac{1}{720}x^6$$

Differentiating with respect to x yields:

$$e'(x) = -\frac{17}{2097152}x - \frac{49}{16777216}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5$$

This is already squarefree, so the squarefree decomposition is trivial. It has fewer real roots than its full complement of 5, just 3 of them, all in the interval of interest. Hence we need to use Sturm's theorem. The Sturm sequence given by Maple, after rescaling the original derivative and all the members of the sequence

[24] Note that the $n = 4$ value indicates that the minimax approximation we are concerned with is 5 times as accurate as the truncated Taylor series of the same order, a difference that suffices to make a significant change to the overall error in the algorithm.

to make all the coefficients integers, can easily be checked in HOL by expanding the definitions and doing arithmetic. We can then deduce from Sturm's theorem that there are exactly 3 real roots in the interval concerned. First, we want to isolate them to a reasonable accuracy. For the accuracy of the polynomial bound to be well above the tightness of the error bound required, we choose $\epsilon = 2^{-48}$. It is straightforward to give a crude upper bound on the derivative using the following theorem:

```
|- ∀x k p. abs x <= k ⟹ abs(poly p x) <= poly (MAP abs p) k
```

and we duly find that $B = 2^{-21}$ suffices. Therefore we want to isolate the roots to within $\epsilon/B = 2^{-27}$. Maple offers the following isolating intervals when given the required accuracy: $[0, 0]$, $[\frac{936399}{2^{27}}, \frac{936400}{2^{27}}]$ and $[\frac{-935680}{2^{27}}, \frac{-935679}{2^{27}}]$.

We can now prove that an ordered list of these isolating intervals does indeed include all the roots, using the following general lemma:

```
|- ∀l a b.
    {x | a <= x ∧ x <= b ∧ (poly p x = &0)} HAS_SIZE (LENGTH l) ∧
    recordered a l b ∧
    FORALL (λ(u,v). poly p(u) * poly p(v) <= &0) l
    ⟹ ∀x. a <= x ∧ x <= b ∧ (poly p(x) = &0)
            ⟹ EX (λ(u,v). u <= x ∧ x <= v) l
```

where:

```
|- (recordered a [] b = a <= b) ∧
   (recordered a (CONS h t) b =
        a < FST h ∧ FST h <= SND h ∧ recordered (SND h) t b)
```

As we said, there is already a theorem in HOL asserting that we can maximize a differentiable function in an interval by considering values only at the endpoints and points of zero derivative:

```
|- ∀f f' a b K.
      (∀x. a <= x ∧ x <= b ⟹ (f diffl (f' x)) x) ∧
      abs(f a) <= K ∧ abs(f b) <= K ∧
      (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0) ⟹ abs(f x) <= K)
      ⟹ (∀x. a <= x ∧ x <= b ⟹ abs(f x) <= K)
```

This can be modified to take account of our approximate knowledge of the points of zero derivative:

```
|- ∀f f' l a b.
    (∀x. a <= x ∧ x <= b ⟹ (f diffl f'(x)) x) ∧
    (∀x. a <= x ∧ x <= b ⟹ abs(f'(x)) <= B) ∧
    abs(f a) <= K + B * e ∧ abs(f b) <= K + B * e ∧
    (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0)
         ⟹ EX (λ(u,v). u <= x ∧ x <= v) l) ∧
    FORALL (λ(u,v). a <= u ∧ v <= b ∧
                    abs(u - v) <= e ∧ abs(f(u)) <= K) l
    ⟹ ∀x. a <= x ∧ x <= b ⟹ abs(f(x)) <= K + B * e
```

We now have all the ingredients required by this theorem, so we get the final result for the error in approximating the higher degree Taylor series. We now include the error from the Taylor truncation, using the following theorem to give a crude bound on the error term:

```
|- ∀x. &0 <= x ∧ x <= inv(&2) ⟹ exp(x) <= &1 + &2 * x
```

Thus we get the final result:

```
|- ∀x. --(&10844) / &1000000 <= x ∧ x <= &10844 / &1000000
       ⟹ abs((exp(x) - &1) - (x + (&8388676 / &2 pow 24) * x pow 2 +
                                  &11184876 / &2 pow 26 * x pow 3))
           <= (&24 / &27) * inv(&2 pow 33)
```

D List of definitions and theorems

Here we collect most of the definitions and theorems that are involved in the work described here, starting with some of the most useful facts of real analysis.

General theorems of real analysis

```
IVT =
 |- ∀f a b y.
        a <= b ∧
        (f a <= y ∧ y <= f b) ∧
        (∀x. a <= x ∧ x <= b ⟹ f contl x)
        ⟹ (∃x. a <= x ∧ x <= b ∧ (f x = y))
IVT2 =
 |- ∀f a b y.
        a <= b ∧
        (f b <= y ∧ y <= f a) ∧
        (∀x. a <= x ∧ x <= b ⟹ f contl x)
        ⟹ (∃x. a <= x ∧ x <= b ∧ (f x = y))
MVT =
 |- ∀f a b.
        a < b ∧
        (∀x. a <= x ∧ x <= b ⟹ f contl x) ∧
        (∀x. a < x ∧ x < b ⟹ f differentiable x)
        ⟹ (∃l z.
                a < z ∧ z < b ∧
                (f diffl l) z ∧ (f b - f a = (b - a) * l))
```

Basic theorems about exponential

```
DIFF_EXP = |- ∀x. (exp diffl exp x) x
REAL_EXP_0 = |- exp(&0) = &1
REAL_EXP_LE_X = |- ∀x. &0 <= x ⟹ &1 + x <= exp x
REAL_EXP_LT_1 = |- ∀x. &0 < x ⟹ &1 < exp x
REAL_EXP_ADD_MUL = |- ∀x y. exp(x + y) * exp(--x) = exp y
REAL_EXP_NEG_MUL = |- ∀x. exp x * exp(--x) = &1
REAL_EXP_NEG_MUL2 = |- ∀x. exp(--x) * exp x = &1
REAL_EXP_NEG = |- ∀x. exp(--x) = inv(exp x)
REAL_EXP_ADD = |- ∀x y. exp(x + y) = exp x * exp y
REAL_EXP_POS_LE = |- ∀x. &0 <= exp x
REAL_EXP_NZ = |- ∀x. ¬(exp x = &0)
REAL_EXP_POS_LT = |- ∀x. &0 < exp x
REAL_EXP_N = |- ∀n x. exp(&n * x) = exp x pow n
REAL_EXP_SUB = |- ∀x y. exp(x - y) = exp x / exp y
REAL_EXP_MONO_IMP = |- ∀x y. x < y ⟹ exp x < exp y
REAL_EXP_MONO_LT = |- ∀x y. exp x < exp y = x < y
REAL_EXP_MONO_LE = |- ∀x y. exp x <= exp y = x <= y
REAL_EXP_INJ = |- ∀x y. (exp x = exp y) = (x = y)
REAL_EXP_TOTAL_LEMMA =
 |- ∀y. &1 <= y ⟹ (∃x. &0 <= x ∧ x <= y - &1 ∧ (exp x = y))
REAL_EXP_TOTAL = |- ∀y. &0 < y ⟹ (∃x. exp x = y)
REAL_EXP_BOUND_LEMMA =
 |- ∀x. &0 <= x ∧ x <= inv(&2) ⟹ exp x <= &1 + &2 * x
REAL_EXP_MINUS1_BOUND_LEMMA =
 |- ∀x. &0 <= x ∧ x <= inv(&2) ⟹ &1 - exp(--x) <= &2 * x
REAL_EXP_SIGN = |- ∀x. (&1 <= exp x = &0 <= x) ∧ (exp x < &1 = x < &0)
REAL_EXP_MINUS1_BOUND =
 |- ∀x. abs x <= inv(&2) ⟹ abs(exp x - &1) <= &2 * abs x
```

Basic theorems about logarithm and roots

```
ln = |- ∀x. ln x = (εu. exp u = x)
LN_EXP = |- ∀x. ln(exp x) = x
REAL_EXP_LN = |- ∀x. (exp(ln x) = x) = &0 < x
LN_MUL = |- ∀x y. &0 < x ∧ &0 < y ⟹ (ln(x * y) = ln x + ln y)
LN_INJ = |- ∀x y. &0 < x ∧ &0 < y ⟹ ((ln x = ln y) = (x = y))
LN_1 = |- ln(&1) = &0
LN_INV = |- ∀x. &0 < x ⟹ (ln(inv x) = --(ln x))
LN_DIV = |- ∀x. &0 < x ∧ &0 < y ⟹ (ln(x / y) = ln x - ln y)
LN_MONO_LT = |- ∀x y. &0 < x ∧ &0 < y ⟹ (ln x < ln y = x < y)
LN_MONO_LE = |- ∀x y. &0 < x ∧ &0 < y ⟹ (ln x <= ln y = x <= y)
LN_POW = |- ∀n x. &0 < x ⟹ (ln(x pow n) = &n * ln x)
LN_LE = |- ∀x. &0 <= x ⟹ ln(&1 + x) <= x
LN_LT_X = |- ∀x. &0 < x ⟹ ln x < x
LN_POS = |- ∀x. &1 <= x ⟹ &0 <= ln x
DIFF_LN = |- ∀x. &0 < x ⟹ (ln diffl inv x) x
LN_1_MINUS_X =
 |- ∀x. &0 < x ∧ x <= inv(&4) ⟹ abs(ln(&1 - x) + x) <= x pow 2
LN_NEAR_1 =
 |- ∀x e. abs(x - &1) <= e ∧ e <= inv(&4) ⟹ abs(ln x) <= e + e pow 2
root = |- ∀n x. root n x = (εu. (&0 < x ⟹ &0 < u) ∧ (u pow n = x))
sqrt = |- ∀x. sqrt x = root 2 x
ROOT_LT_LEMMA =
 |- ∀n x. &0 < x ⟹ (exp(ln x / &(SUC n)) pow SUC n = x)
ROOT_LN =
 |- ∀x. &0 < x ⟹ (∀n. root(SUC n) x = exp(ln x / &(SUC n)))
ROOT_0 = |- ∀n. root(SUC n) (&0) = &0
ROOT_1 = |- ∀n. root(SUC n) (&1) = &1
ROOT_POW_POS = |- ∀n x. &0 <= x ⟹ (root(SUC n) x pow SUC n = x)
POW_ROOT_POS = |- ∀n x. &0 <= x ⟹ (root(SUC n) (x pow SUC n) = x)
ROOT_POS_POSITIVE = |- ∀x n. &0 <= x ⟹ &0 <= root(SUC n) x
SQRT_0 = |- sqrt(&0) = &0
SQRT_1 = |- sqrt(&1) = &1
SQRT_POW2 = |- ∀x. (sqrt x pow 2 = x) = &0 <= x
```

Taylor/Mclaurin expansions

```
MCLAURIN =
 |- ∀f diff h n.
        &0 < h ∧
        0 < n ∧
        (diff 0 = f) ∧
        (∀m t.
             m < n ∧ &0 <= t ∧ t <= h ⟹ (diff m diffl diff (SUC m) t) t)
        ⟹ (∃t. &0 < t ∧
                t < h ∧
                (f h =
                 Sum (0,n) (λm. diff m (&0) / &(FACT m) * h pow m) +
                 diff n t / &(FACT n) * h pow n))
MCLAURIN_NEG =
 |- ∀f diff h n.
        h < &0 ∧
        0 < n ∧
        (diff 0 = f) ∧
        (∀m t.
             m < n ∧ h <= t ∧ t <= &0 ⟹ (diff m diffl diff (SUC m) t) t)
        ⟹ (∃t. h < t ∧
                t < &0 ∧
                (f h =
                 Sum (0,n) (λm. diff m (&0) / &(FACT m) * h pow m) +
                 diff n t / &(FACT n) * h pow n))
MCLAURIN_ALL_LT =
 |- ∀f diff.
        (diff 0 = f) ∧ (∀m x. (diff m diffl diff (SUC m) x) x)
        ⟹ (∀x n.
                ¬(x = &0) ∧ 0 < n
                ⟹ (∃t. &0 < abs t ∧
                        abs t < abs x ∧
                        (f x =
                         Sum (0,n)
                          (λm. diff m (&0) / &(FACT m) * x pow m) +
                         diff n t / &(FACT n) * x pow n)))
MCLAURIN_ALL_LE =
 |- ∀f diff.
        (diff 0 = f) ∧ (∀m x. (diff m diffl diff (SUC m) x) x)
        ⟹ (∀x n.
                ∃t. abs t <= abs x ∧
                    (f x =
                     Sum (0,n) (λm. diff m (&0) / &(FACT m) * x pow m) +
                     diff n t / &(FACT n) * x pow n))
```

Mclaurin series for exponential and logarithm

```
MCLAURIN_EXP_LT =
 |- ∀x n.
        ¬(x = &0) ∧ 0 < n
        ⟹ (∃t. &0 < abs t ∧
                abs t < abs x ∧
                (exp x =
                 Sum (0,n) (λm. x pow m / &(FACT m)) +
                 exp t / &(FACT n) * x pow n))
MCLAURIN_EXP_LE =
 |- ∀x n.
        ∃t. abs t <= abs x ∧
            (exp x =
             Sum (0,n) (λm. x pow m / &(FACT m)) +
             exp t / &(FACT n) * x pow n)
MCLAURIN_LN_POS =
 |- ∀x n.
        &0 < x ∧ x < &1 ∧ 0 < n
        ⟹ (∃t. &0 < t ∧
                t < x ∧
                (ln(&1 + x) =
                 Sum (0,n) (λm. --(&1) pow SUC m * x pow m / &m) +
                 --(&1) pow SUC n * x pow n / (&n * (&1 + t) pow n)))
MCLAURIN_LN_NEG =
 |- ∀x n.
        &0 < x ∧ x < &1 ∧ 0 < n
        ⟹ (∃t. &0 < t ∧
                t < x ∧
                (--(ln(&1 - x)) =
                 Sum (0,n) (λm. x pow m / &m) +
                 x pow n / (&n * (&1 - t) pow n)))
```

Basic definitions and theorems for polynomials

```
poly = |- (poly [] x = &0) ∧ (poly (CONS h t) x = h + x * poly t x)
poly_add =
 |- ([] ++ l2 = l2) ∧
    (CONS h t ++ l2 =
     (if l2 = [] then CONS h t else CONS (h + HD l2) (t ++ TL l2)))
poly_cmul =
 |- (c ## [] = []) ∧ (c ## CONS h t = CONS (c * h) (c ## t))
poly_neg = |- neg = $poly_cmul (--(&1))
poly_mul =
 |- ([] ** l2 = []) ∧
    (CONS h t ** l2 =
     (if t = [] then h ## l2 else h ## l2 ++ CONS (&0) (t ** l2)))
poly_exp = |- (p exp 0 = [&1]) ∧ (p exp SUC n = p ** p exp n)
poly_diff_aux =
 |- (poly_diff_aux n [] = []) ∧
    (poly_diff_aux n (CONS h t) = CONS (&n * h) (poly_diff_aux (SUC n) t))
POLY_ADD_CLAUSES =
 |- ([] ++ p2 = p2) ∧
    (p1 ++ [] = p1) ∧
    (CONS h1 t1 ++ CONS h2 t2 = CONS (h1 + h2) (t1 ++ t2))
POLY_CMUL_CLAUSES =
 |- (c ## [] = []) ∧ (c ## CONS h t = CONS (c * h) (c ## t))
POLY_NEG_CLAUSES =
 |- (neg [] = []) ∧ (neg (CONS h t) = CONS (--h) (neg t))
POLY_MUL_CLAUSES =
 |- ([] ** p2 = []) ∧
    ([h1] ** p2 = h1 ## p2) ∧
    (CONS h1 (CONS k1 t1) ** p2 = h1 ## p2 ++ CONS (&0) (CONS k1 t1 ** p2))
POLY_DIFF_CLAUSES =
 |- (diff [] = []) ∧
    (diff [c] = []) ∧
    (diff (CONS h t) = poly_diff_aux 1 t)
POLY_ADD = |- ∀p1 p2 x. poly (p1 ++ p2) x = poly p1 x + poly p2 x
POLY_CMUL = |- ∀p c x. poly (c ## p) x = c * poly p x
POLY_NEG = |- ∀p x. poly (neg p) x = --(poly p x)
POLY_MUL = |- ∀x p1 p2. poly (p1 ** p2) x = poly p1 x * poly p2 x
POLY_EXP = |- ∀p n x. poly (p exp n) x = poly p x pow n
```

Analytical properties of polynomials

```
poly_diff =
 |- ∀l. diff l = (if l = [] then [] else poly_diff_aux 1 (TL l))
POLY_DIFF = |- ∀l x. ((λx. poly l x) diffl poly (diff l) x) x
POLY_DIFFERENTIABLE = |- ∀l x. (λx. poly l x) differentiable x
POLY_CONT = |- ∀l x. (λx. poly l x) contl x
POLY_IVT_POS =
 |- ∀p a b.
        a < b ∧ poly p a < &0 ∧ poly p b > &0
        ⟹ (∃x. a < x ∧ x < b ∧ (poly p x = &0))
POLY_IVT_NEG =
 |- ∀p a b.
        a < b ∧ poly p a > &0 ∧ poly p b < &0
        ⟹ (∃x. a < x ∧ x < b ∧ (poly p x = &0))
POLY_MVT =
 |- ∀p a b.
        a < b
        ⟹ (∃x. a < x ∧
                x < b ∧
                (poly p b - poly p a = (b - a) * poly (diff p) x))
```

Lemmas about polynomials (1)

```
POLY_ADD_RZERO = |- ∀p. poly (p ++ []) = poly p
POLY_MUL_ASSOC = |- ∀p q r. poly (p ** q ** r) = poly ((p ** q) ** r)
POLY_EXP_ADD =
 |- ∀d n p. poly (p exp(n + d)) = poly (p exp n ** p exp d)
POLY_DIFF_AUX_ADD =
 |- ∀p1 p2 n.
        poly (poly_diff_aux n (p1 ++ p2)) =
        poly (poly_diff_aux n p1 ++ poly_diff_aux n p2)
POLY_DIFF_AUX_CMUL =
 |- ∀p c n. poly (poly_diff_aux n (c ## p)) =
            poly (c ## poly_diff_aux n p)
POLY_DIFF_AUX_NEG =
 |- ∀p n. poly (poly_diff_aux n (neg p)) = poly (neg (poly_diff_aux n p))
POLY_DIFF_AUX_MUL_LEMMA =
 |- ∀p n. poly (poly_diff_aux (SUC n) p) = poly (poly_diff_aux n p ++ p)
POLY_DIFF_ADD =
 |- ∀p1 p2. poly (diff (p1 ++ p2)) = poly (diff p1 ++ diff p2)
POLY_DIFF_CMUL = |- ∀p c. poly (diff (c ## p)) = poly (c ## diff p)
POLY_DIFF_NEG = |- ∀p. poly (diff (neg p)) = poly (neg (diff p))
POLY_DIFF_MUL_LEMMA =
 |- ∀t h. poly (diff (CONS h t)) = poly (CONS (&0) (diff t) ++ t)
POLY_DIFF_MUL =
 |- ∀p1 p2. poly (diff (p1 ** p2)) = poly (p1 ** diff p2 ++ diff p1 ** p2)
POLY_DIFF_EXP =
 |- ∀p n. poly (diff (p exp SUC n)) =
          poly ((&(SUC n) ## p exp n) ** diff p)
POLY_DIFF_EXP_PRIME =
 |- ∀n a.
        poly (diff ([--a; &1] exp SUC n)) =
        poly (&(SUC n) ## [--a; &1] exp n)
POLY_LINEAR_REM = |- ∀t h. ∃q r. CONS h t = [r] ++ [--a; &1] ** q
POLY_LINEAR_DIVIDES =
 |- ∀a p. (poly p a = &0) = (p = []) ∨ (∃q. p = [--a; &1] ** q)
POLY_LENGTH_MUL = |- ∀q. LENGTH ([--a; &1] ** q) = SUC (LENGTH q)
POLY_ROOTS_INDEX_LEMMA =
 |- ∀n p.
        ¬(poly p = poly []) ∧ (LENGTH p = n)
        ⟹ (∃i. ∀x. (poly p x = &0) ⟹ (∃m. m <= n ∧ (x = i m)))
POLY_ROOTS_INDEX_LENGTH =
 |- ∀p. ¬(poly p = poly [])
        ⟹ (∃i. ∀x. (poly p x = &0) ⟹ (∃n. n <= LENGTH p ∧ (x = i n)))
POLY_ROOTS_FINITE_LEMMA =
 |- ∀p. ¬(poly p = poly [])
        ⟹ (∃N i. ∀x. (poly p x = &0) ⟹ (∃n. n < N ∧ (x = i n)))
FINITE_LEMMA =
 |- ∀i N P.
        (∀x. P x ⟹ (∃n. n < N ∧ (x = i n))) ⟹ (∃a. ∀x. P x ⟹ x < a)
POLY_ROOTS_FINITE =
 |- ∀p. ¬(poly p = poly []) =
        (∃N i. ∀x. (poly p x = &0) ⟹ (∃n. n < N ∧ (x = i n)))
```

Lemmas about polynomials (2)

```
POLY_ENTIRE_LEMMA =
 |- ∀p q.
        ¬(poly p = poly []) ∧ ¬(poly q = poly [])
        ⟹ ¬(poly (p ** q) = poly [])
POLY_ENTIRE =
 |- ∀p q.
        (poly (p ** q) = poly []) =
        (poly p = poly []) ∨ (poly q = poly [])
POLY_MUL_LCANCEL =
 |- ∀p q r.
        (poly (p ** q) = poly (p ** r)) =
        (poly p = poly []) ∨ (poly q = poly r)
POLY_EXP_EQ_0 =
 |- ∀p n. (poly (p exp n) = poly []) = (poly p = poly []) ∧ ¬(n = 0)
POLY_PRIME_EQ_0 = |- ∀a. ¬(poly [a; &1] = poly [])
POLY_EXP_PRIME_EQ_0 = |- ∀a n. ¬(poly ([a; &1] exp n) = poly [])
POLY_ZERO_LEMMA =
 |- ∀h t. (poly (CONS h t) = poly []) ⟹ (h = &0) ∧ (poly t = poly [])
POLY_ZERO = |- ∀p. (poly p = poly []) = FORALL (λc. c = &0) p
POLY_DIFF_AUX_ISZERO =
 |- ∀p n.
        FORALL (λc. c = &0) (poly_diff_aux (SUC n) p) =
        FORALL (λc. c = &0) p
POLY_DIFF_ISZERO =
 |- ∀p. (poly (diff p) = poly []) ⟹ (∃h. poly p = poly [h])
POLY_DIFF_ZERO =
 |- ∀p. (poly p = poly []) ⟹ (poly (diff p) = poly [])
POLY_DIFF_WELLDEF =
 |- ∀p q. (poly p = poly q) ⟹ (poly (diff p) = poly (diff q))
```

Divisibility properties of polynomials

```
divides = |- ∀p2 p1. p1 divides p2 = (∃q. poly p2 = poly (p1 ** q))
POLY_PRIMES =
 |- ∀a p q. [a; &1] divides p ** q = [a; &1] divides p ∨ [a; &1] divides q
POLY_DIVIDES_REFL = |- ∀p. p divides p
POLY_DIVIDES_TRANS =
 |- ∀p q r. p divides q ∧ q divides r ⟹ p divides r
POLY_DIVIDES_EXP = |- ∀p m n. m <= n ⟹ p exp m divides p exp n
POLY_EXP_DIVIDES =
 |- ∀p q m n. p exp n divides q ∧ m <= n ⟹ p exp m divides q
POLY_DIVIDES_ADD =
 |- ∀p q r. p divides q ∧ p divides r ⟹ p divides q ++ r
POLY_DIVIDES_SUB =
 |- ∀p q r. p divides q ∧ p divides q ++ r ⟹ p divides r
POLY_DIVIDES_SUB2 =
 |- ∀p q r. p divides r ∧ p divides q ++ r ⟹ p divides q
POLY_DIVIDES_ZERO = |- ∀p q. (poly p = poly []) ⟹ q divides p
```

Orders of roots

```
POLY_ORDER_EXISTS =
 |- ∀a d p.
        (LENGTH p = d) ∧ ¬(poly p = poly [])
        ⟹ (∃n. [--a; &1] exp n divides p ∧
                ¬([--a; &1] exp SUC n divides p))
POLY_ORDER =
 |- ∀p a.
        ¬(poly p = poly [])
        ⟹ (∃∀n. [--a; &1] exp n divides p ∧
                 ¬([--a; &1] exp SUC n divides p))
order =
 |- ∀a p.
        order a p =
        (εn. [--a; &1] exp n divides p ∧ ¬([--a; &1] exp SUC n divides p))
ORDER =
 |- ∀p a n.
        [--a; &1] exp n divides p ∧ ¬([--a; &1] exp SUC n divides p) =
        (n = order a p) ∧ ¬(poly p = poly [])
ORDER_THM =
 |- ∀p a.
        ¬(poly p = poly [])
        ⟹ [--a; &1] exp order a p divides p ∧
            ¬([--a; &1] exp SUC (order a p) divides p)
ORDER_UNIQUE =
 |- ∀p a n.
        ¬(poly p = poly []) ∧
        [--a; &1] exp n divides p ∧
        ¬([--a; &1] exp SUC n divides p)
        ⟹ (n = order a p)
ORDER_POLY = |- ∀p q a. (poly p = poly q) ⟹ (order a p = order a q)
ORDER_ROOT =
 |- ∀p a. (poly p a = &0) = (poly p = poly []) ∨ ¬(order a p = 0)
ORDER_DIVIDES =
 |- ∀p a n. [--a; &1] exp n divides p =
            (poly p = poly []) ∨ n <= order a p
ORDER_DECOMP =
 |- ∀p a.
        ¬(poly p = poly [])
        ⟹ (∃q. (poly p = poly ([--a; &1] exp order a p ** q)) ∧
                ¬([--a; &1] divides q))
ORDER_MUL =
 |- ∀a p q.
        ¬(poly (p ** q) = poly [])
        ⟹ (order a (p ** q) = order a p + order a q)
ORDER_DIFF =
 |- ∀p a.
        ¬(poly (diff p) = poly []) ∧ ¬(order a p = 0)
        ⟹ (order a p = SUC (order a (diff p)))
```

Squarefree decomposition and normalization

```
POLY_SQUAREFREE_DECOMP_ORDER =
 |- ∀p q d e r s.
        ¬(poly (diff p) = poly []) ∧
        (poly p = poly (q ** d)) ∧
        (poly (diff p) = poly (e ** d)) ∧
        (poly d = poly (r ** p ++ s ** diff p))
        ⟹ (∀a. order a q = (if order a p = 0 then 0 else 1))
rsquarefree =
 |- ∀p. rsquarefree p =
        ¬(poly p = poly []) ∧ (∀a. (order a p = 0) ∨ (order a p = 1))
RSQUAREFREE_ROOTS =
 |- ∀p. rsquarefree p = (∀a. ¬((poly p a = &0) ∧ (poly (diff p) a = &0)))
RSQUAREFREE_DECOMP =
 |- ∀p a.
        rsquarefree p ∧ (poly p a = &0)
        ⟹ (∃q. (poly p = poly ([--a; &1] ** q)) ∧ ¬(poly q a = &0))
POLY_SQUAREFREE_DECOMP =
 |- ∀p q d e r s.
        ¬(poly (diff p) = poly []) ∧
        (poly p = poly (q ** d)) ∧
        (poly (diff p) = poly (e ** d)) ∧
        (poly d = poly (r ** p ++ s ** diff p))
        ⟹ rsquarefree q ∧ (∀a. (poly q a = &0) = (poly p a = &0))
normalize =
 |- (normalize [] = []) ∧
    (normalize (CONS h t) =
     (if normalize t = []
      then if h = &0 then [] else [h]
      else CONS h (normalize t)))
POLY_NORMALIZE = |- ∀p. poly (normalize p) = poly p
degree = |- ∀p. degree p = PRE (LENGTH (normalize p))
DEGREE_ZERO = |- ∀p. (poly p = poly []) ⟹ (degree p = 0)
POLY_ROOTS_FINITE_SET =
 |- ∀p. ¬(poly p = poly []) ⟹ FINITE {x | poly p x = &0}
POLY_MONO =
 |- ∀x k p. abs x <= k ⟹ abs(poly p x) <= poly (MAP abs p) k
```

Separation of roots

```
POLYS_ROOTS_FINITE_SET =
 |- ∀l. FORALL (λp. ¬(poly p = poly [])) l
        ⟹ FINITE {x | EX (λp. poly p x = &0) l}
POLYS_INTERVAL_ROOTS_FINITE_SET =
 |- ∀l a b.
        FORALL (λp. ¬(poly p = poly [])) l
        ⟹ FINITE {x | a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) l}
FINITE_LEAST =
 |- ∀s. FINITE s ⟹ (s = EMPTY) ∨ (∃a. a IN s ∧ (∀x. x IN s ⟹ a <= x))
FINITE_LEAST_DELETE =
 |- ∀s. s HAS_SIZE SUC n
        ⟹ (∃a. a IN s ∧
                 s DELETE a HAS_SIZE n ∧
                 (∀x. x IN s DELETE a ⟹ a < x))
HAS_SIZE_ORDER =
 |- ∀n s.
        s HAS_SIZE n
        ⟹ (∃i. (∀x. x IN s = (∃k. k < n ∧ (x = i k))) ∧
                 (∀k. i k < i (SUC k)))
FINITE_ORDER =
 |- ∀s. FINITE s
        ⟹ (∃i N.
                 (∀x. x IN s = (∃k. k < N ∧ (x = i k))) ∧
                 (∀k. i k < i (SUC k)))
POLYS_ENUM_ROOTS =
 |- ∀l. FORALL (λp. ¬(poly p = poly [])) l
        ⟹ (∃i N.
                 (∀k. i k < i (SUC k)) ∧
                 (∀x. a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) l =
                      (∃n. n < N ∧ (x = i n))))
POLYS_INTERVAL_SEPARATION =
 |- ∀f l a b.
        a <= b ∧
        FORALL (λp. ¬(poly p = poly [])) l ∧
        ¬(poly f a = &0) ∧
        ¬(poly f b = &0)
        ⟹ (∃i N.
                 (i 0 = a) ∧
                 (i N = b) ∧
                 (∀k. i k <= i (SUC k)) ∧
                 (∀k. k <= N ⟹ ¬(poly f (i k) = &0)) ∧
                 (∀k. ∃c. i k <= c ∧
                          c <= i (SUC k) ∧
                          (∀x. i k <= x ∧
                               x <= i (SUC k) ∧
                               EX (λp. poly p x = &0) (CONS f l)
                               ⟹ (x = c))))
```

Lemmas for Sturm's theorem (1)

```
SIGN_LEMMA0 =
 |- ∀a b c.
        &0 < a ∧ &0 < b ∧ &0 < c ⟹ &0 < a * b ∧ &0 < a * c ∧ &0 < b * c
SIGN_LEMMA1 = |- ∀a b c. a * b > &0 ⟹ (c * a < &0 = c * b < &0)
SIGN_LEMMA2 = |- ∀a b c. a * b > &0 ⟹ (a * c < &0 = b * c < &0)
SIGN_LEMMA3 = |- ∀a b. a < &0 ⟹ (a * b > &0 = b < &0)
SIGN_LEMMA4 =
 |- ∀a b c. a * b < &0 ∧ ¬(c = &0) ⟹ (c * a < &0 = ¬(c * b < &0))
SIGN_LEMMA5 = |- a * b < &0 = a > &0 ∧ b < &0 ∨ a < &0 ∧ b > &0
SIGN_LEMMA6 = |- a * b <= &0 = a >= &0 ∧ b <= &0 ∨ a <= &0 ∧ b >= &0
varrec =
 |- (varrec prev [] = 0) ∧
    (varrec prev (CONS h t) =
     (if prev * h < &0
      then SUC (varrec h t)
      else if h = &0 then varrec prev t else varrec h t))
variation = |- ∀l. variation l = varrec (&0) l
VARREC_SIGN = |- ∀l r s. r * s > &0 ⟹ (varrec r l = varrec s l)
VARREC_STRADDLE =
 |- ∀f g h x.
        poly f x * poly h x < &0
        ⟹ (varrec (poly f x) (MAP (λp. poly p x) (CONS g (CONS h t))) =
             SUC (varrec (poly h x) (MAP (λp. poly p x) t)))
STURM =
 |- (STURM f f' [] = f' divides f) ∧
    (STURM f f' (CONS g gs) =
     (∃k. &0 < k ∧ f' divides f ++ k ## g) ∧
     degree g < degree f' ∧
     STURM f' g gs)
STURM_NOROOT =
 |- ∀a b p.
        a <= b ∧ (∀x. a <= x ∧ x <= b ⟹ ¬(poly p x = &0))
        ⟹ poly p a * poly p b > &0
STURM_NOROOT_NOVAR_LEMMA =
 |- ∀f a b l.
        a <= b ∧ (∀x. a <= x ∧ x <= b ⟹ ¬(poly f x = &0))
        ⟹ (varrec r (MAP (λp. poly p a) (CONS f l)) +
             varrec (poly f a) (MAP (λp. poly p b) l) =
             varrec r (MAP (λp. poly p b) (CONS f l)) +
             varrec (poly f a) (MAP (λp. poly p a) l))
STURM_NOROOT_NOVAR =
 |- ∀f a b l.
        a <= b ∧
        (∀x. a <= x ∧ x <= b ⟹ ¬(poly f x = &0)) ∧
        (varrec (poly f a) (MAP (λp. poly p a) l) =
         varrec (poly f a) (MAP (λp. poly p b) l))
        ⟹ (varrec r (MAP (λp. poly p a) (CONS f l)) =
             varrec r (MAP (λp. poly p b) (CONS f l)))
```

Lemmas for Sturm's theorem (2)

```
STURM_NOVAR_LEMMA =
 |- ∀n l f f' c.
        (LENGTH l = n) ∧
        STURM f f' l ∧
        a <= c ∧
        c <= b ∧
        (∀x. a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) (CONS f (CONS f' l))
             ⟹ (x = c)) ∧
        ¬(poly f c = &0)
        ⟹ (varrec (poly f a) (MAP (λp. poly p a) (CONS f' l)) =
             varrec (poly f b) (MAP (λp. poly p b) (CONS f' l)))
STURM_NOVAR =
 |- ∀l f f' c.
        STURM f f' l ∧
        a <= c ∧
        c <= b ∧
        (∀x. a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) (CONS f (CONS f' l))
             ⟹ (x = c)) ∧
        ¬(poly f c = &0)
        ⟹ (varrec (poly f a) (MAP (λp. poly p a) (CONS f' l)) =
             varrec (poly f b) (MAP (λp. poly p b) (CONS f' l)))
STURM_VAR_DLEMMA =
 |- ∀p a b.
        a <= b ∧ (∀x. a <= x ∧ x <= b ⟹ ¬(poly p x = &0))
        ⟹ (∀x. a <= x ∧ x <= b ⟹ poly p x > &0) ∨
            (∀x. a <= x ∧ x <= b ⟹ poly p x < &0)
STURM_VAR_LEMMA =
 |- ∀l f f' c.
        rsquarefree f ∧
        (f' = diff f) ∧
        STURM f f' l ∧
        a < c ∧
        c < b ∧
        (∀x. a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) (CONS f (CONS f' l))
             ⟹ (x = c)) ∧
        (poly f c = &0)
        ⟹ poly f a * poly f' a < &0 ∧ poly f b * poly f' b > &0
STURM_VAR =
 |- ∀l f f' c.
        rsquarefree f ∧
        (f' = diff f) ∧
        STURM f f' l ∧
        a < c ∧
        c < b ∧
        (∀x. a <= x ∧ x <= b ∧ EX (λp. poly p x = &0) (CONS f (CONS f' l))
             ⟹ (x = c)) ∧
        (poly f c = &0)
        ⟹ (varrec (poly f a) (MAP (λp. poly p a) (CONS f' l)) =
             SUC (varrec (poly f b) (MAP (λp. poly p b) (CONS f' l))))
```

Weak forms of Sturm's theorem

```
STURM_COMPONENT =
 |- ∀l f a b c.
     rsquarefree f ∧
     STURM f (diff f) l ∧
     a <= c ∧
     c <= b ∧
     ¬(poly f a = &0) ∧
     ¬(poly f b = &0) ∧
     (∀x. a <= x ∧
          x <= b ∧
          EX (λp. poly p x = &0) (CONS f (CONS (diff f) l))
          ⟹ (x = c))
     ⟹ (variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))) =
          (if poly f c = &0
           then SUC
                (variation
                 (MAP (λp. poly p b) (CONS f (CONS (diff f) l))))
           else variation
                 (MAP (λp. poly p b) (CONS f (CONS (diff f) l)))))
STURM_LEMMA =
 |- ∀i n.
     rsquarefree f ∧
     STURM f (diff f) l ∧
     (∀k. i k <= i (SUC k)) ∧
     (∀k. k <= n ⟹ ¬(poly f (i k) = &0)) ∧
     (∀k. ∃c. i k <= c ∧
              c <= i (SUC k) ∧
              (∀x. i k <= x ∧
                   x <= i (SUC k) ∧
                   EX (λp. poly p x = &0) (CONS f (CONS (diff f) l))
                   ⟹ (x = c)))
     ⟹ FINITE {x | i 0 <= x ∧ x <= i n ∧ (poly f x = &0)} ∧
         (variation (MAP (λp. poly p (i n)) (CONS f (CONS (diff f) l))) +
          CARD {x | i 0 <= x ∧ x <= i n ∧ (poly f x = &0)} =
          variation (MAP (λp. poly p (i 0)) (CONS f (CONS (diff f) l))))
STURM_NONZERO_LEMMA =
 |- ∀l f f'. ¬(poly f = poly []) ∧ STURM f f' l ⟹ ¬(poly f' = poly [])
STURM_NONZERO =
 |- ∀l f f'.
        ¬(poly f = poly []) ∧ STURM f f' l
        ⟹ FORALL (λp. ¬(poly p = poly [])) (CONS f' l)
```

Stronger forms of Sturm's theorem

```
STURM_STRONG =
 |- ∀f a b l.
        a <= b ∧
        ¬(poly f a = &0) ∧
        ¬(poly f b = &0) ∧
        rsquarefree f ∧
        STURM f (diff f) l
        ⟹ FINITE {x | a <= x ∧ x <= b ∧ (poly f x = &0)} ∧
            (variation (MAP (λp. poly p b) (CONS f (CONS (diff f) l))) +
             CARD {x | a <= x ∧ x <= b ∧ (poly f x = &0)} =
             variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))))
STURM_THM =
 |- ∀f a b l.
        a <= b ∧
        ¬(poly f a = &0) ∧
        ¬(poly f b = &0) ∧
        rsquarefree f ∧
        STURM f (diff f) l
        ⟹ {x | a <= x ∧ x <= b ∧ (poly f x = &0)} HAS_SIZE
            variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))) -
            variation (MAP (λp. poly p b) (CONS f (CONS (diff f) l)))
STURM_GCD =
 |- ∀l k f f'.
        STURM f f' (CONS k l)
        ⟹ (∃q e r s.
                (poly f = poly (q ** LAST (CONS k l))) ∧
                (poly f' = poly (e ** LAST (CONS k l))) ∧
                (poly (LAST (CONS k l)) = poly (r ** f ++ s ** f')))
STURM_THEOREM =
 |- ∀f a b l d.
        a <= b ∧
        ¬(poly f a = &0) ∧
        ¬(poly f b = &0) ∧
        ¬(poly (diff f) = poly []) ∧
        STURM f (diff f) l ∧
        ¬(l = []) ∧
        (LAST l = [d]) ∧
        ¬(d = &0)
        ⟹ {x | a <= x ∧ x <= b ∧ (poly f x = &0)} HAS_SIZE
            variation (MAP (λp. poly p a) (CONS f (CONS (diff f) l))) -
            variation (MAP (λp. poly p b) (CONS f (CONS (diff f) l)))
```

Bounding differentiable functions

```
BOUND_THEOREM_POS =
 |- ∀f f' a b K.
        (∀x. a <= x ∧ x <= b ⟹ (f diffl f' x) x) ∧
        f a <= K ∧
        f b <= K ∧
        (∀x. a <= x ∧ x <= b ∧ (f' x = &0) ⟹ f x <= K)
        ⟹ (∀x. a <= x ∧ x <= b ⟹ f x <= K)
BOUND_THEOREM_NEG =
 |- ∀f f' a b K.
        (∀x. a <= x ∧ x <= b ⟹ (f diffl f' x) x) ∧
        K <= f a ∧
        K <= f b ∧
        (∀x. a <= x ∧ x <= b ∧ (f' x = &0) ⟹ K <= f x)
        ⟹ (∀x. a <= x ∧ x <= b ⟹ K <= f x)
BOUND_THEOREM_EXACT =
 |- ∀f f' a b K.
        (∀x. a <= x ∧ x <= b ⟹ (f diffl f' x) x) ∧
        abs(f a) <= K ∧
        abs(f b) <= K ∧
        (∀x. a <= x ∧ x <= b ∧ (f' x = &0) ⟹ abs(f x) <= K)
        ⟹ (∀x. a <= x ∧ x <= b ⟹ abs(f x) <= K)
BOUND_THEOREM_EXACT_ALT =
 |- ∀f f' a b K.
        (∀x. a <= x ∧ x <= b ⟹ (f diffl f' x) x) ∧
        (∀x. (x = a) ∨ (x = b) ∨ a < x ∧ x < b ∧ (f' x = &0)
             ⟹ abs(f x) <= K)
        ⟹ (∀x. a <= x ∧ x <= b ⟹ abs(f x) <= K)
BOUND_THEOREM_INEXACT =
 |- ∀f f' l a b.
        (∀x. a <= x ∧ x <= b ⟹ (f diffl f' x) x) ∧
        (∀x. a <= x ∧ x <= b ⟹ abs(f' x) <= B) ∧
        abs(f a) <= K + B * e ∧
        abs(f b) <= K + B * e ∧
        (∀x. a <= x ∧ x <= b ∧ (f' x = &0)
             ⟹ EX (λ(u,v). u <= x ∧ x <= v) l) ∧
        FORALL (λ(u,v). a <= u ∧ v <= b ∧
                        abs(u - v) <= e ∧ abs(f u) <= K) l
        ⟹ (∀x. a <= x ∧ x <= b ⟹ abs(f x) <= K + B * e)
```

Bounding in terms of ordering derivative's roots

```
recordered =
 |- (recordered a [] b = a <= b) ∧
    (recordered a (CONS h t) b =
     a < FST h ∧ FST h <= SND h ∧ recordered (SND h) t b)
RECORDERED_CLAUSES =
 |- (recordered a [] b = a <= b) ∧
    (recordered a (CONS (u,v) t) b = a < u ∧ u <= v ∧ recordered v t b)
RECORDERED_MONO =
 |- ∀l a a' b. a <= a' ∧ recordered a' l b ⟹ recordered a l b
FINITE_AD_HOC_LEMMA =
 |- ∀a b n.
        (a INTER b = EMPTY) ∧ (∃x. x IN a) ∧ a UNION b HAS_SIZE n
        ⟹ (∃m. m < n ∧ b HAS_SIZE m)
RECORDERED_CONTAINED_LEMMA =
 |- ∀l a b n.
        recordered a l b ∧
        FORALL (λ(u,v). poly p u * poly p v <= &0) l ∧
        {x | a <= x ∧
             x <= b ∧
             (poly p x = &0) ∧
             EX (λ(u,v). u <= x ∧ x <= v) l} HAS_SIZE
        n
        ⟹ LENGTH l <= n
RECORDERED_ROOTCOUNT =
 |- ∀l a b.
        {x | a <= x ∧ x <= b ∧ (poly p x = &0)} HAS_SIZE LENGTH l ∧
        recordered a l b ∧
        FORALL (λ(u,v). poly p u * poly p v <= &0) l
        ⟹ (∀x. a <= x ∧ x <= b ∧ (poly p x = &0)
                ⟹ EX (λ(u,v). u <= x ∧ x <= v) l)
SINGLE_DERIVBOUND =
 |- ∀x. abs x <= &10844 / &1000000
        ⟹ abs
            (poly
             [&0; --(&17) / &2097152; --(&49) / &16777216; &1 / &6;
             &1 /
             &24; &1 / &120]
            x) <=
            inv(&2 pow 21)
```

Tang's single precision polynomial approximation

```
SINGLE_STURM =
 |- STURM [&0; --(&2040); --(&735); &41943040; &10485760; &2097152]
         (poly_diff [&0; --(&2040); --(&735); &41943040;
                     &10485760; &2097152])
         [[--(&408); &1338; &25166265; --(&8388608)];
          [&12549417544; --(&21760245838); --(&774072258592059)];
          [--(&39740632656867464012); --(&4415519623031413035448627)];
          [--(&1)]]
SINGLE_ROOTCOUNT_LEMMA =
 |- ∀a b.
       a <= b ∧
       ¬(poly [&0; --(&2040); --(&735); &41943040; &10485760; &2097152]
                     a = &0) ∧
       ¬(poly [&0; --(&2040); --(&735); &41943040; &10485760; &2097152]
                     b = &0)
       ⟹ {x | a <= x ∧
                x <= b ∧
                (poly
                 [&0; --(&2040); --(&735); &41943040; &10485760; &2097152]
                 x = &0)} HAS_SIZE
       (variation (MAP (λp. poly p a)
         [[&0; --(&2040); --(&735); &41943040; &10485760; &2097152];
          [--(&2040); --(&1470); &125829120; &41943040; &10485760];
          [--(&408); &1338; &25166265; --(&8388608)];
          [&12549417544; --(&21760245838); --(&774072258592059)];
          [--(&39740632656867464012); --(&4415519623031413035448627)];
          [--(&1)]]) -
        variation (MAP (λp. poly p b)
          [[&0; --(&2040); --(&735); &41943040; &10485760; &2097152];
          [--(&2040); --(&1470); &125829120; &41943040; &10485760];
          [--(&408); &1338; &25166265; --(&8388608)];
          [&12549417544; --(&21760245838); --(&774072258592059)];
          [--(&39740632656867464012); --(&4415519623031413035448627)];
          [--(&1)]]))
SINGLE_ROOTCOUNT_1_UNSCALED =
 |- {x | --(&1) <= x ∧
         x <= &1 ∧
         (poly [&0; --(&2040); --(&735); &41943040; &10485760; &2097152]
               x = &0)} HAS_SIZE 3
SINGLE_ROOTCOUNT_1 =
 |- {x | --(&1) <= x ∧
         x <= &1 ∧
         (poly
          [&0; --(&17) / &2097152; --(&49) / &16777216; &1 / &6; &1 / &24;
           &1 / &120] x = &0)} HAS_SIZE 3
```

Accuracy theorems for polynomial approximation

```
SINGLE_ROOTLIST_1 =
 |- ∀x. --(&1) <= x ∧
        x <= &1 ∧
        (poly
         [&0; --(&17) / &2097152; --(&49) / &16777216; &1 / &6; &1 / &24;
          &1 / &120] x = &0)
        ⟹ EX (λ(u,v). u <= x ∧ x <= v)
            [(--(&935680) / &2 pow 27,--(&935679) / &2 pow 27); (&0,&0);
            (&936399 / &2 pow 27, &936400 / &2 pow 27)]
SINGLE_ROOTLIST =
 |- ∀x. --(&10844) / &1000000 <= x ∧
        x <= &10844 / &1000000 ∧
        (poly
         [&0; --(&17) / &2097152; --(&49) / &16777216; &1 / &6; &1 / &24;
          &1 / &120] x = &0)
        ⟹ EX (λ(u,v). u <= x ∧ x <= v)
            [(--(&935680) / &2 pow 27,--(&935679) / &2 pow 27); (&0,&0);
            (&936399 / &2 pow 27, &936400 / &2 pow 27)]
SINGLE_ACCURACY_LEMMA =
 |- ∀x. --(&10844) / &1000000 <= x ∧ x <= &10844 / &1000000
        ⟹ abs
            (poly
             [&0; &0; --(&17) / &4194304; --(&49) / &50331648; &1 / &24;
             &1 /
             &120; &1 / &720]
            x) <=
            &1835023447399279096794056781021 1 /
            &184320000000000000000000000000000000000000 +
            inv(&2 pow 21) * inv(&2 pow 27)
SINGLE_MCLAURIN_ACCURACY_LEMMA =
 |- ∀t. t <= &10844 / &1000000
        ⟹ abs(exp t) <= &1 + &2 * &10844 / &1000000
SINGLE_MCLAURIN_ACCURACY =
 |- ∀x t.
        abs t <= abs x ∧ abs x <= &10844 / &1000000
        ⟹ abs(exp t * inv(&5040) * x pow 7) <= inv(&2 pow 57)
SINGLE_ACCURACY =
 |- ∀x. --(&10844) / &1000000 <= x ∧ x <= &10844 / &1000000
        ⟹ abs
            (exp x -
             &1 -
             (x +
              &8388676 / &2 pow 24 * x pow 2 +
              &11184876 / &2 pow 26 * x pow 3)) <=
            &24 / &27 * inv(&2 pow 33)
```

Basic programming language definitions

```
False = |- False = (λx. F)
True = |- True = (λx. T)
Not = |- ∀p. Not p = (λx. ¬p x)
And = |- ∀p q. p And q = (λx. p x ∧ q x)
Or = |- ∀p q. p Or q = (λx. p x ∨ q x)
Ands = |- ∀P. Ands P = (λx. ∀p. P p ⟹ p x)
Ors = |- ∀P. Ors P = (λx. ∃p. P p ∧ p x)
Imp = |- ∀p q. p Imp q = (λx. p x ⟹ q x)
Implies = |- ∀p q. p Implies q = (∀x. p x ⟹ q x)
PRED_TAUT : term -> thm = <fun>
IMPLIES_TRANS = |- ∀p q r. p Implies q ∧ q Implies r ⟹ p Implies r
command_INDUCTION =
 |- ∀P. (∀a. P (Assign a)) ∧
        (∀a0 a1. P a0 ∧ P a1 ⟹ P (a0 Seq a1)) ∧
        (∀a0 a1 a2. P a1 ∧ P a2 ⟹ P (Ite a0 a1 a2)) ∧
        (∀a0 a1. P a1 ⟹ P (While a0 a1))
        ⟹ (∀x. P x)
command_RECURSION =
 |- ∀Assign' Seq' Ite' While'.
     ∃fn. (∀a. fn (Assign a) = Assign' a) ∧
          (∀a0 a1. fn (a0 Seq a1) = Seq' a0 a1 (fn a0) (fn a1)) ∧
          (∀a0 a1 a2. fn (Ite a0 a1 a2) = Ite' a0 a1 a2 (fn a1) (fn a2)) ∧
          (∀a0 a1. fn (While a0 a1) = While' a0 a1 (fn a1))
Skip = |- Skip = Assign I
Abort = |- Abort = While True Skip
If = |- ∀e c. If e c = Ite e c Skip
Do = |- ∀e c. Do c e = c Seq While e c
Assert = |- ∀g. Assert g = Ite g Skip Abort
Awhile = |- ∀i v e c. Awhile i v e c = While e c
Ado = |- ∀i v c e. Ado i v c e = Do c e
command_DISTINCT =
 |- (∀a a0' a1'. ¬(Assign a = a0' Seq a1')) ∧
    (∀a a0' a1' a2'. ¬(Assign a = Ite a0' a1' a2')) ∧
    (∀a a0' a1'. ¬(Assign a = While a0' a1')) ∧
    (∀a0 a1 a0' a1' a2'. ¬(a0 Seq a1 = Ite a0' a1' a2')) ∧
    (∀a0 a1 a0' a1'. ¬(a0 Seq a1 = While a0' a1')) ∧
    (∀a0 a1 a2 a0' a1'. ¬(Ite a0 a1 a2 = While a0' a1'))
command_INJECTIVE =
 |- (∀a a'. (Assign a = Assign a') = (a = a')) ∧
    (∀a0 a1 a0' a1'. (a0 Seq a1 = a0' Seq a1') =
                     (a0 = a0') ∧ (a1 = a1')) ∧
    (∀a0 a1 a2 a0' a1' a2'.
         (Ite a0 a1 a2 = Ite a0' a1' a2') =
         (a0 = a0') ∧ (a1 = a1') ∧ (a2 = a2')) ∧
    (∀a0 a1 a0' a1'.
         (While a0 a1 = While a0' a1') = (a0 = a0') ∧ (a1 = a1'))
```

Simple semantic theorems

```
sem_RULES =
 |- (∀f s. sem (Assign f) s (f s)) ∧
    (∀c1 c2 s s' s''. sem c1 s s' ∧ sem c2 s' s''
                      ⟹ sem (c1 Seq c2) s s'') ∧
    (∀e c1 c2 s s'. e s ∧ sem c1 s s' ⟹ sem (Ite e c1 c2) s s') ∧
    (∀e c1 c2 s s'. ¬e s ∧ sem c2 s s' ⟹ sem (Ite e c1 c2) s s') ∧
    (∀e c s. ¬e s ⟹ sem (While e c) s s) ∧
    (∀e c s s' s''.
         e s ∧ sem c s s' ∧ sem (While e c) s' s''
         ⟹ sem (While e c) s s'')
sem_INDUCT =
 |- ∀sem'. (∀f s. sem' (Assign f) s (f s)) ∧
           (∀c1 c2 s s' s''.
              sem' c1 s s' ∧ sem' c2 s' s'' ⟹ sem' (c1 Seq c2) s s'') ∧
           (∀e c1 c2 s s'.
              e s ∧ sem' c1 s s' ⟹ sem' (Ite e c1 c2) s s') ∧
           (∀e c1 c2 s s'. ¬e s ∧ sem' c2 s s'
                           ⟹ sem' (Ite e c1 c2) s s') ∧
           (∀e c s. ¬e s ⟹ sem' (While e c) s s) ∧
           (∀e c s s' s''.
                e s ∧ sem' c s s' ∧ sem' (While e c) s' s''
                ⟹ sem' (While e c) s s'')
           ⟹ (∀a0 a1 a2. sem a0 a1 a2 ⟹ sem' a0 a1 a2)
sem_CASES =
 |- ∀a0 a1 a2.
        sem a0 a1 a2 =
        (∃f. (a0 = Assign f) ∧ (a2 = f a1)) ∨
        (∃c1 c2 s'. (a0 = c1 Seq c2) ∧ sem c1 a1 s' ∧ sem c2 s' a2) ∨
        (∃e c1 c2. (a0 = Ite e c1 c2) ∧ e a1 ∧ sem c1 a1 a2) ∨
        (∃e c1 c2. (a0 = Ite e c1 c2) ∧ ¬e a1 ∧ sem c2 a1 a2) ∨
        (∃e c. (a0 = While e c) ∧ (a2 = a1) ∧ ¬e a1) ∨
        (∃e c s'.
             (a0 = While e c) ∧ e a1 ∧
             sem c a1 s' ∧ sem (While e c) s' a2)
SEM_ASSIGN = |- sem (Assign f) s s' = (s' = f s)
SEM_SEQ = |- sem (c1 Seq c2) s s' = (∃s''. sem c1 s s'' ∧ sem c2 s'' s')
SEM_ITE =
 |- sem (Ite e c1 c2) s s' = e s ∧ sem c1 s s' ∨ ¬e s ∧ sem c2 s s'
SEM_SKIP = |- sem Skip s s' = (s' = s)
SEM_IF = |- sem (If e c) s s' = e s ∧ sem c s s' ∨ ¬e s ∧ (s = s')
SEM_WHILE = |- sem (While e c) s s' = sem (If e (c Seq While e c)) s s'
SEM_ABORT = |- sem Abort s s' = F
SEM_DO = |- sem (Do c e) s s' = sem (c Seq If e (Do c e)) s s'
SEM_ASSERT = |- sem (Assert g) s s' = g s ∧ (s' = s)
```

Weakest preconditions etc.

```
deterministic =
 |- ∀r. deterministic r = (∀s s1 s2. r s s1 ∧ r s s2 ⟹ (s1 = s2))
DETERMINISM = |- ∀c. deterministic (sem c)
terminates = |- ∀c s. terminates c s = (∃s'. sem c s s')
wlp = |- ∀c s q. wlp c q s = (∀s'. sem c s s' ⟹ q s')
wp = |- ∀c q s. wp c q s = terminates c s ∧ wlp c q s
WP_TOTAL = |- ∀c. wp c False = False
WP_MONOTONIC = |- q Implies r ⟹ wp c q Implies wp c r
WP_CONJUNCTIVE = |- wp c q And wp c r = wp c (q And r)
WP_DISJUNCTIVE = |- wp c p Or wp c q = wp c (p Or q)
WP_ASSIGN = |- ∀f q. wp (Assign f) q = q o f
WP_SEQ = |- ∀c1 c2 q. wp (c1 Seq c2) q = wp c1 (wp c2 q)
WP_ITE =
 |- ∀e c1 c2 q. wp (Ite e c1 c2) q = e And wp c1 q Or Not e And wp c2 q
WP_WHILE = |- ∀e c. wp (If e (c Seq While e c)) q = wp (While e c) q
WP_SKIP = |- ∀q. wp Skip q = q
WP_ABORT = |- ∀q. wp Abort q = False
WP_IF = |- ∀e c q. wp (If e c) q = e And wp c q Or Not e And q
WP_DO = |- ∀e c. wp (c Seq If e (Do c e)) q = wp (Do c e) q
WP_ASSERT = |- ∀g q. wp (Assert g) q = g And q
```

Hoare rules for total correctness

```
correct = |- ∀p c q. correct p c q = p Implies wp c q
CORRECT_PRESTRENGTH =
 |- ∀p p' c q. p Implies p' ∧ correct p' c q ⟹ correct p c q
CORRECT_POSTWEAK =
 |- ∀p c q q'. correct p c q' ∧ q' Implies q ⟹ correct p c q
CORRECT_ASSIGN =
 |- ∀p f q. p Implies (λs. q (f s)) ⟹ correct p (Assign f) q
CORRECT_SEQ =
 |- ∀p q r c1 c2. correct p c1 r ∧ correct r c2 q
                            ⟹ correct p (c1 Seq c2) q
CORRECT_ITE =
 |- ∀p e c1 c2 q.
        correct (p And e) c1 q ∧ correct (p And Not e) c2 q
        ⟹ correct p (Ite e c1 c2) q
CORRECT_WHILE =
 |- ∀<< p c q e invariant.
        WF $<< ∧
        p Implies invariant ∧
        Not e And invariant Implies q ∧
        (∀X. correct (invariant And e And (λs. X = s)) c
             (invariant And (λs. s << X)))
        ⟹ correct p (While e c) q
CORRECT_SKIP = |- ∀p q. p Implies q ⟹ correct p Skip q
CORRECT_ABORT = |- ∀p q. F ⟹ correct p Abort q
CORRECT_IF =
 |- ∀p e c q.
        correct (p And e) c q ∧ p And Not e Implies q
        ⟹ correct p (If e c) q
CORRECT_DO =
 |- ∀<< p q c invariant.
        WF $<< ∧
        e And invariant Implies p ∧
        Not e And invariant Implies q ∧
        (∀X. correct (p And (λs. X = s)) c (invariant And (λs. s << X)))
        ⟹ correct p (Do c e) q
CORRECT_ASSERT =
 |- ∀p g q. p Implies g And q ⟹ correct p (Assert g) q
```

Theorems to drive VC generation

```
VC_ASSIGN = |- p Implies q o f ⟹ correct p (Assign f) q
VC_SKIP = |- p Implies q ⟹ correct p Skip q
VC_ABORT = |- F ⟹ correct p Abort q
VC_ASSERT = |- p Implies b And q ⟹ correct p (Assert b) q
VC_ITE =
 |- correct (p And e) c1 q ∧ correct (p And Not e) c2 q
    ⟹ correct p (Ite e c1 c2) q
VC_IF =
 |- correct (p And e) c q ∧ p And Not e Implies q ⟹ correct p (If e c) q
VC_AWHILE_VARIANT =
 |- WF $<< ∧
    p Implies invariant ∧
    Not e And invariant Implies q ∧
    (∀X. correct (invariant And e And (λs. X = s)) c
         (invariant And (λs. s << X)))
    ⟹ correct p (Awhile invariant $<< e c) q
VC_AWHILE_MEASURE =
 |- p Implies invariant ∧
    Not e And invariant Implies q ∧
    (∀X. correct (invariant And e And (λs. X = m s)) c
         (invariant And (λs. m s < X)))
    ⟹ correct p (Awhile invariant (measure m) e c) q
VC_ADO_VARIANT =
 |- WF $<< ∧
    e And invariant Implies p ∧
    Not e And invariant Implies q ∧
    (∀X. correct (p And (λs. X = s)) c (invariant And (λs. s << X)))
    ⟹ correct p (Ado invariant $<< c e) q
VC_ADO_MEASURE =
 |- e And invariant Implies p ∧
    Not e And invariant Implies q ∧
    (∀X. correct (p And (λs. X = m s)) c (invariant And (λs. m s < X)))
    ⟹ correct p (Ado invariant (measure m) c e) q
VC_SEQ_ASSERT_LEFT =
 |- p Implies b ∧ correct b c q ⟹ correct p (Assert b Seq c) q
VC_SEQ_ASSERT_RIGHT =
 |- correct p c b ∧ b Implies q ⟹ correct p (c Seq Assert b) q
VC_SEQ_ASSERT_MIDDLE =
 |- correct p c b ∧ correct b c' q ⟹ correct p (c Seq Assert b Seq c') q
VC_SEQ_ASSIGN_LEFT =
 |- (p o f = p) ∧ (f o f = f) ∧ correct (p And (λs. s = f s)) c q
    ⟹ correct p (Assign f Seq c) q
VC_SEQ_ASSIGN_RIGHT =
 |- correct p c (q o f) ⟹ correct p (c Seq Assign f) q
```

Basic definitions for general IEEE arithmetic

```
expwidth = |- ∀fw ew. expwidth (ew,fw) = ew
fracwidth = |- ∀ew fw. fracwidth (ew,fw) = fw
wordlength = |- ∀X. wordlength X = expwidth X + fracwidth X + 1
emax = |- ∀X. emax X = 2 EXP expwidth X - 1
bias = |- ∀X. bias X = 2 exp(expwidth X - 1) - 1
is_single =
 |- ∀X. is_single X = (expwidth X = 8) ∧ (wordlength X = 32)
is_double =
 |- ∀X. is_double X = (expwidth X = 11) ∧ (wordlength X = 64)
is_single_extended =
 |- ∀X. is_single_extended X = expwidth X >= 11 ∧ wordlength X >= 43
is_double_extended =
 |- ∀X. is_double_extended X = expwidth X >= 15 ∧ wordlength X >= 79
sign = |- ∀e f s. sign (s,(e,f)) = s
exponent = |- ∀s f e. exponent (s,(e,f)) = e
fraction = |- ∀s e f. fraction (s,(e,f)) = f
is_nan =
 |- ∀X a. is_nan X a = (exponent a = emax X) ∧ ¬(fraction a = 0)
is_infinity =
 |- ∀X a. is_infinity X a = (exponent a = emax X) ∧ (fraction a = 0)
is_normal =
 |- ∀a X. is_normal X a = 0 < exponent a ∧ exponent a < emax X
is_denormal =
 |- ∀X a. is_denormal X a = (exponent a = 0) ∧ ¬(fraction a = 0)
is_zero = |- ∀X a. is_zero X a = (exponent a = 0) ∧ (fraction a = 0)
is_valid =
 |- ∀s e f X.
        is_valid X (s,(e,f)) =
        s < 2 ∧ e < 2 EXP expwidth X ∧ f < 2 EXP fracwidth X
is_finite =
 |- ∀X a.
        is_finite X a =
        is_valid X a ∧ (is_normal X a ∨ is_denormal X a ∨ is_zero X a)
plus_infinity = |- ∀X. plus_infinity X = 0,(emax X,0)
minus_infinity = |- ∀X. minus_infinity X = 1,(emax X,0)
plus_zero = |- ∀X. plus_zero X = 0,(0,0)
minus_zero = |- ∀X. minus_zero X = 1,(0,0)
topfloat = |- ∀X. topfloat X = 0,(emax X - 1,2 EXP fracwidth X - 1)
bottomfloat =
 |- ∀X. bottomfloat X = 1,(emax X - 1,2 EXP fracwidth X - 1)
minus = |- ∀X a. minus X a = 1 - sign a,(exponent a,fraction a)
```

Valuation, useful values, rounding modes

```
encoding =
 |- ∀s e X f.
        encoding X (s,(e,f)) =
        s * 2 exp(wordlength X - 1) + e * 2 EXP fracwidth X + f
valof =
 |- ∀s e f X.
        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))
largest =
 |- ∀X. largest X =
        &2 pow (emax X - 1) / &2 pow bias X *
        (&2 - inv(&2 pow fracwidth X))
threshold =
 |- ∀X. threshold X =
        &2 pow (emax X - 1) / &2 pow bias X *
        (&2 - inv(&2 pow SUC (fracwidth X)))
ulp =
 |- ∀X a. ulp X a =
          valof X (0,(exponent a,1)) - valof X (0,(exponent a,0))
some_nan = |- ∀X. some_nan X = (εa. is_nan X a)
roundmode_INDUCT =
 |- ∀P. P To_nearest ∧ P To_zero ∧ P To_pinfinity ∧ P To_ninfinity
        ⟹ (∀x. P x)
roundmode_RECURSION =
 |- ∀To_nearest' To_zero' To_pinfinity' To_ninfinity'.
        ∃fn. (fn To_nearest = To_nearest') ∧
             (fn To_zero = To_zero') ∧
             (fn To_pinfinity = To_pinfinity') ∧
             (fn To_ninfinity = To_ninfinity')
```

Rounding to floating point values

```
is_closest =
 |- ∀s a v x.
        is_closest v s x a =
        a IN s ∧ (∀b. b IN s ⟹ abs(v a - x) <= abs(v b - x))
closest =
 |- ∀v s x p.
        closest v p s x =
        (εa. is_closest v s x a ∧
             ((∃b. is_closest v s x b ∧ p b) ⟹ p a))
round =
 |- (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)) ∧
    (round X To_zero x =
     (if x < --(largest X)
      then bottomfloat X
      else if x > largest X
           then topfloat X
           else closest (valof X) (λx. T)
                {a | is_finite X a ∧ abs(valof X a) <= abs x}
                x)) ∧
    (round X To_pinfinity x =
     (if x < --(largest X)
      then bottomfloat X
      else if x > largest X
           then plus_infinity X
           else closest (valof X) (λx. T)
                {a | is_finite X a ∧ valof X a >= x}
                x)) ∧
    (round X To_ninfinity x =
     (if x < --(largest X)
      then minus_infinity X
      else if x > largest X
           then topfloat X
           else closest (valof X) (λx. T)
                {a | is_finite X a ∧ valof X a <= x}
                x))
```

Rounding to integers

```
is_integral =
 |- ∀X a. is_integral X a = is_finite X a ∧ (∃n. abs(valof X a) = &n)
intround =
 |- (intround 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. ∃n. EVEN n ∧ (abs(valof X a) = &n))
                         {a | is_integral X a}
                x)) ∧
    (intround X To_zero x =
     (if x < --(largest X)
      then bottomfloat X
      else if x > largest X
           then topfloat X
           else closest (valof X) (λx. T)
                {a | is_integral X a ∧ abs(valof X a) <= abs x}
                x)) ∧
    (intround X To_pinfinity x =
     (if x < --(largest X)
      then bottomfloat X
      else if x > largest X
           then plus_infinity X
           else closest (valof X) (λx. T)
                {a | is_integral X a ∧ valof X a >= x}
                x)) ∧
    (intround X To_ninfinity x =
     (if x < --(largest X)
      then minus_infinity X
      else if x > largest X
           then topfloat X
           else closest (valof X) (λx. T)
                {a | is_integral X a ∧ valof X a <= x}
                x))
```

Arithmetic operations (1)

```
zerosign =
 |- ∀s X a.
        zerosign X s a =
        (if is_zero X a
         then if s = 0 then plus_zero X else minus_zero X
         else a)
rem =
 |- ∀x y.
        x rem y =
        let n =
            closest I (λx. ∃n. EVEN n ∧ (abs x = &n)) {x | ∃n. abs x = &n}
            (x / y) in x - n * y
fintrnd =
 |- ∀m X a.
        fintrnd X m a =
        (if is_nan X a
         then some_nan X
         else if is_infinity X a
              then a
              else zerosign X (sign a) (intround X m (valof X a)))
fadd =
 |- ∀m a X b.
        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)))
fsub =
 |- ∀m a X b.
        fsub 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 minus X 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)))
```

Arithmetic operations (2)

```
fmul =
 |- ∀m a X b.
        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)))
fdiv =
 |- ∀m a X b.
        fdiv X m a b =
        (if is_nan X a ∨
            is_nan X b ∨
            is_zero X a ∧ is_zero X b ∨
            is_infinity X a ∧ is_infinity X b
         then some_nan X
         else if is_infinity X a ∨ is_zero X b
              then if sign a = sign b
                   then plus_infinity X
                   else minus_infinity X
              else if is_infinity X b
                   then if sign a = sign b then plus_zero X
                        else minus_zero X
                   else zerosign X (if sign a = sign b then 0 else 1)
                        (round X m (valof X a / valof X b)))
fsqrt =
 |- ∀m X a.
        fsqrt X m a =
        (if is_nan X a
         then some_nan X
         else if is_zero X a ∨ is_infinity X a ∧ (sign a = 0)
              then a
              else if sign a = 1
                   then some_nan X
                   else zerosign X (sign a) (round X m (sqrt(valof X a))))
frem =
 |- ∀m a X b.
        frem X m a b =
        (if is_nan X a ∨ is_nan X b ∨ is_infinity X a ∨ is_zero X b
         then some_nan X
         else if is_infinity X b
              then a
              else zerosign X (sign a)
                     (round X m (valof X a rem valof X b)))

fneg = |- ∀X m a. fneg X m a = 1 - sign a,(exponent a,fraction a)
```

Comparison operations

```
ccode_INDUCTION = |- ∀P. P Gt ∧ P Lt ∧ P Eq ∧ P Un ⟹ (∀x. P x)
ccode_RECURSION =
 |- ∀Gt' Lt' Eq' Un'.
       ∃fn. (fn Gt = Gt') ∧ (fn Lt = Lt') ∧ (fn Eq = Eq') ∧ (fn Un = Un')
fcompare =
 |- ∀a X b.
        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)
flt = |- ∀X a b. flt X a b = (fcompare X a b = Lt)
fle =
 |- ∀X a b. fle X a b = (fcompare X a b = Lt) ∨ (fcompare X a b = Eq)
fgt = |- ∀X a b. fgt X a b = (fcompare X a b = Gt)
fge =
 |- ∀X a b. fge X a b = (fcompare X a b = Gt) ∨ (fcompare X a b = Eq)
feq = |- ∀X a b. feq X a b = (fcompare X a b = Eq)
```

Float format

```
float_format = |- float_format = 8,23
float_tybij =
 |- (∀a. float (defloat a) = a) ∧
    (∀r. is_valid float_format r = (defloat (float r) = r))
Val = |- ∀a. Val a = valof float_format (defloat a)
Float = |- ∀x. Float x = float (round float_format To_nearest x)
Sign = |- ∀a. Sign a = sign (defloat a)
Exponent = |- ∀a. Exponent a = exponent (defloat a)
Fraction = |- ∀a. Fraction a = fraction (defloat a)
Ulp = |- ∀a. Ulp a = ulp float_format (defloat a)
Isnan = |- ∀a. Isnan a = is_nan float_format (defloat a)
Infinity = |- ∀a. Infinity a = is_infinity float_format (defloat a)
Isnormal = |- ∀a. Isnormal a = is_normal float_format (defloat a)
Isdenormal = |- ∀a. Isdenormal a = is_denormal float_format (defloat a)
Iszero = |- ∀a. Iszero a = is_zero float_format (defloat a)
Finite = |- ∀a. Finite a = Isnormal a ∨ Isdenormal a ∨ Iszero a
Isintegral = |- ∀a. Isintegral a = is_integral float_format (defloat a)
Plus_zero = |- Plus_zero = float (plus_zero float_format)
Minus_zero = |- Minus_zero = float (minus_zero float_format)
Minus_infinity =
 |- Minus_infinity = float (minus_infinity float_format)
Plus_infinity = |- Plus_infinity = float (plus_infinity float_format)
add_f =
 |- ∀a b.
      a + b = float (fadd float_format To_nearest (defloat a) (defloat b))
sub_f =
 |- ∀a b.
      a - b = float (fsub float_format To_nearest (defloat a) (defloat b))
mul_f =
 |- ∀a b.
      a * b = float (fmul float_format To_nearest (defloat a) (defloat b))
div_f =
 |- ∀a b.
      a / b = float (fdiv float_format To_nearest (defloat a) (defloat b))
rem_f =
 |- ∀a b.
      a % b = float (frem float_format To_nearest (defloat a) (defloat b))
sqrt_f =
 |- ∀a. sqrt a = float (fsqrt float_format To_nearest (defloat a))
ROUNDFLOAT =
 |- ∀a. ROUNDFLOAT a = float (fintrnd float_format To_nearest (defloat a))
lt_f = |- ∀a b. a < b = flt float_format (defloat a) (defloat b)
le_f = |- ∀a b. a <= b = fle float_format (defloat a) (defloat b)
gt_f = |- ∀a b. a > b = fgt float_format (defloat a) (defloat b)
ge_f = |- ∀a b. a >= b = fge float_format (defloat a) (defloat b)
eq_f = |- ∀a b. a == b = feq float_format (defloat a) (defloat b)
neg_f = |- ∀a. --a = float (fneg float_format To_nearest (defloat a))
abs_f = |- ∀a. abs a = (if a >= Plus_zero then a else --a)
```

Definitions of machine integers

```
Int_tybij =
 |- (∀a. Int (Inum a) = a) ∧ (∀r. r < 2 EXP 32 = (Inum (Int r) = r))
Ival =
 |- ∀a. Ival a =
        (if Inum a < 2 EXP 31 then &(Inum a) else &(Inum a) - &(2 EXP 32))
Icoerce = |- ∀x. Icoerce x = (εa. Ival a = x)
INT_DIFFERENCE_LEMMA = |- ∀a b. ∃k. abs(&a - &b) = &k
INT_IVAL_LEMMA = |- ∀i. ∃n. abs(Ival i) = &n
Int_real_tybij =
 |- (∀a. Icoerce (Ival a) = a) ∧
    (∀r. (∃n. abs r = &n) ∧ --(&2 pow 31) <= r ∧ r < &2 pow 31 =
         (Ival (Icoerce r) = r))
Int_add = |- ∀a b. a + b = Icoerce (Ival a + Ival b)
Int_sub = |- ∀a b. a - b = Icoerce (Ival a - Ival b)
Int_mul = |- ∀a b. a * b = Icoerce (Ival a * Ival b)
Int_divmod =
 |- ∀a b.
        Int_divmod a b =
        (εdm. &0 <= Ival (SND dm) ∧
              Ival (SND dm) < abs(Ival b) ∧
              (Ival a = Ival (FST dm) * Ival b + Ival (SND dm)))
Int_div = |- ∀a b. a / b = FST (Int_divmod a b)
Int_mod = |- ∀a b. a % b = SND (Int_divmod a b)
Int_neg = |- ∀a. --a = Icoerce (--(Ival a))
Int_lt = |- ∀a b. a < b = Ival a < Ival b
Int_le = |- ∀a b. a <= b = Ival a <= Ival b
Int_gt = |- ∀a b. a > b = Ival a > Ival b
Int_ge = |- ∀a b. a >= b = Ival a >= Ival b
Int_eq = |- ∀a b. a == b = (Ival a = Ival b)
Int_abs = |- ∀a. abs a = (if a >= Int 0 then a else --a)
fscalb =
 |- ∀m N X a.
        fscalb X m (a,N) =
        (if is_nan X a
         then some_nan X
         else if is_infinity X a
              then a
              else zerosign X (sign a)
                   (round X m (exp(Ival N * ln(&2)) * valof X a)))
Scalb =
 |- ∀a N. Scalb (a,N) =
           float (fscalb float_format To_nearest (defloat a,N))
Toint = |- ∀a. Toint a = (εi. Ival i = Val a)
Tofloat = |- ∀i. Tofloat i = (εa. Finite a ∧ (Val a = Ival i))
INTRND = |- ∀a. INTRND a = Toint (ROUNDFLOAT a)
```

Miscellaneous lemmas about integers

```
INT_DIVMOD_LEMMA =
 |- ∀a b.
        (∃m. abs a = &m) ∧ (∃n. abs b = &n) ∧ ¬(b = &0)
        ⟹ (∃q r.
                (∃p. abs q = &p) ∧
                (∃s. abs r = &s) ∧
                &0 <= r ∧
                r < abs b ∧
                abs q <= abs a ∧
                (a = q * b + r))
INT_DIVMOD_EXISTS =
 |- ∀a b.
        ¬(Ival b = &0) ∧ --(&2 pow 31) < Ival a
        ⟹ (∃q r.
                &0 <= Ival r ∧
                Ival r < abs(Ival b) ∧
                (Ival a = Ival q * Ival b + Ival r))
INT_DIVMOD_UNIQUE =
 |- ∀a b q1 r1 q2 r2.
        &0 <= Ival r1 ∧
        Ival r1 < Ival b ∧
        (Ival a = Ival q1 * Ival b + Ival r1) ∧
        &0 <= Ival r2 ∧
        Ival r2 < Ival b ∧
        (Ival a = Ival q2 * Ival b + Ival r2)
        ⟹ (q1 = q2) ∧ (r1 = r2)
INT_DIVMOD =
 |- ∀a b.
        ¬(Ival b = &0) ∧ --(&2 pow 31) < Ival a
        ⟹ &0 <= Ival (a % b) ∧
            Ival (a % b) < abs(Ival b) ∧
            (Ival a = Ival (a / b) * Ival b + Ival (a % b))
IVAL_INT_LEMMA = |- ∀n. n < 2 EXP 31 ⟹ (Ival (Int n) = &n)
INT_NEG =
 |- ∀a. ¬(Ival a = --(&2 pow 31)) ⟹ (Ival (--a) = --(Ival a))
INT_ABS =
 |- ∀a. ¬(Ival a = --(&2 pow 31)) ⟹ (Ival (abs a) = abs(Ival a))
```

Convenient variants of definitions

```
SIGN = |- ∀a. sign a = FST a
EXPONENT = |- ∀a. exponent a = FST (SND a)
FRACTION = |- ∀a. fraction a = SND (SND a)
IS_VALID =
 |- ∀X a.
        is_valid X a =
        sign a < 2 ∧
        exponent a < 2 EXP expwidth X ∧
        fraction a < 2 EXP fracwidth X
VALOF =
 |- ∀X a.
        valof X a =
        (if exponent a = 0
         then --(&1) pow sign a *
              &2 / &2 pow bias X *
              &(fraction a) / &2 pow fracwidth X
         else --(&1) pow sign a *
              &2 pow exponent a / &2 pow bias X *
              (&1 + &(fraction a) / &2 pow fracwidth X))
```

Miscellaneous lemmas about floats

```
IS_VALID_DEFLOAT = |- ∀a. is_valid float_format (defloat a)
IS_FINITE_ALT =
 |- ∀a. is_finite float_format a =
        is_valid float_format a ∧ exponent a < 255
IS_FINITE_EXPLICIT =
 |- ∀a. is_finite float_format a =
        sign a < 2 ∧ exponent a < 255 ∧ fraction a < 8388608
FLOAT_CASES =
 |- ∀a. Isnan a ∨ Infinity a ∨ Isnormal a ∨ Isdenormal a ∨ Iszero a
FLOAT_CASES_FINITE = |- ∀a. Isnan a ∨ Infinity a ∨ Finite a
FLOAT_DISTINCT =
 |- ∀a. ¬(Isnan a ∧ Infinity a) ∧
        ¬(Isnan a ∧ Isnormal a) ∧
        ¬(Isnan a ∧ Isdenormal a) ∧
        ¬(Isnan a ∧ Iszero a) ∧
        ¬(Infinity a ∧ Isnormal a) ∧
        ¬(Infinity a ∧ Isdenormal a) ∧
        ¬(Infinity a ∧ Iszero a) ∧
        ¬(Isnormal a ∧ Isdenormal a) ∧
        ¬(Isnormal a ∧ Iszero a) ∧
        ¬(Isdenormal a ∧ Iszero a)
FLOAT_DISTINCT_FINITE =
 |- ∀a. ¬(Isnan a ∧ Infinity a) ∧
        ¬(Isnan a ∧ Finite a) ∧
        ¬(Infinity a ∧ Finite a)
FLOAT_INFINITIES_SIGNED =
 |- (sign (defloat Plus_infinity) = 0) ∧
    (sign (defloat Minus_infinity) = 1)
INFINITY_IS_INFINITY =
 |- Infinity Plus_infinity ∧ Infinity Minus_infinity
ZERO_IS_ZERO = |- Iszero Plus_zero ∧ Iszero Minus_zero
INFINITY_NOT_NAN = |- ¬Isnan Plus_infinity ∧ ¬Isnan Minus_infinity
ZERO_NOT_NAN = |- ¬Isnan Plus_zero ∧ ¬Isnan Minus_zero
FLOAT_INFINITIES =
 |- ∀a. Infinity a = a == Plus_infinity ∨ a == Minus_infinity
FLOAT_INFINITES_DISTINCT =
 |- ∀a. ¬(a == Plus_infinity ∧ a == Minus_infinity)
```

Useful theorems about comparisons

```
FLOAT_LT = |- ∀a b. Finite a ∧ Finite b ⟹ (a < b = Val a < Val b)
FLOAT_GT = |- ∀a b. Finite a ∧ Finite b ⟹ (a > b = Val a > Val b)
FLOAT_LE = |- ∀a b. Finite a ∧ Finite b ⟹ (a <= b = Val a <= Val b)
FLOAT_GE = |- ∀a b. Finite a ∧ Finite b ⟹ (a >= b = Val a >= Val b)
FLOAT_EQ = |- ∀a b. Finite a ∧ Finite b ⟹ (a == b = (Val a = Val b))
FLOAT_EQ_REFL = |- ∀a. a == a = ¬Isnan a
```

Lemmas for analyzing the rounding operation

```
IS_VALID_SPECIAL =
 |- ∀X. is_valid X (minus_infinity X) ∧
        is_valid X (plus_infinity X) ∧
        is_valid X (topfloat X) ∧
        is_valid X (bottomfloat X) ∧
        is_valid X (plus_zero X) ∧
        is_valid X (minus_zero X)
IS_CLOSEST_EXISTS =
 |- ∀v x s. FINITE s ⟹ ¬(s = EMPTY) ⟹ (∃a. is_closest v s x a)
CLOSEST_IS_EVERYTHING =
 |- ∀v p x s.
        FINITE s
        ⟹ ¬(s = EMPTY)
        ⟹ is_closest v s x (closest v p s x) ∧
            ((∃b. is_closest v s x b ∧ p b) ⟹ p (closest v p s x))
CLOSEST_IN_SET =
 |- ∀v p x s. FINITE s ⟹ ¬(s = EMPTY) ⟹ closest v p s x IN s
CLOSEST_IS_CLOSEST =
 |- ∀v p x s.
        FINITE s ⟹ ¬(s = EMPTY) ⟹ is_closest v s x (closest v p s x)
FINITE_R3 =
 |- ∀m n p. FINITE {a | FST a < m ∧ FST (SND a) < n ∧ SND (SND a) < p}
IS_VALID_FINITE = |- FINITE {a | is_valid X a}
IS_FINITE_FINITE = |- ∀X. FINITE {a | is_finite X a}
IS_VALID_NONEMPTY = |- ¬(a | is_valid X a = EMPTY)
IS_FINITE_NONEMPTY = |- ¬(a | is_finite X a = EMPTY)
IS_FINITE_CLOSEST =
 |- ∀X P v p x. is_finite X (closest v p {a | is_finite X a} x)
IS_VALID_CLOSEST =
 |- ∀X P v p x. is_valid X (closest v p {a | is_finite X a} x)
IS_VALID_ROUND = |- ∀X x. is_valid X (round X To_nearest x)
DEFLOAT_FLOAT_ROUND =
 |- ∀X x. defloat (float (round float_format To_nearest x)) =
              round float_format To_nearest x
DEFLOAT_FLOAT_ZEROSIGN_ROUND =
 |- ∀x b.
        defloat (float (zerosign float_format b
           (round float_format To_nearest x))) =
        zerosign float_format b (round float_format To_nearest x)
VALOF_DEFLOAT_FLOAT_ZEROSIGN_ROUND =
 |- ∀x b.
     valof float_format
     (defloat (float (zerosign float_format b
         (round float_format To_nearest x)))) =
     valof float_format (round float_format To_nearest x)
DEFLOAT_FLOAT_ZEROSIGN_ROUND_FINITE =
 |- ∀b x.
     abs x < threshold float_format
     ⟹ is_finite float_format
         (defloat
         (float
         (zerosign float_format b (round float_format To_nearest x))))
```

Useful floating point lemmas

```
ISFINITE = |- ∀a. Finite a = is_finite float_format (defloat a)
VAL_FINITE = |- ∀a. Finite a ⟹ abs(Val a) <= largest float_format
ISFINITE_LEMMA =
 |- ∀s e f.
        s < 2 ∧ e < 255 ∧ f < 2 EXP 23
        ⟹ Finite (float (s,(e,f))) ∧ is_valid float_format (s,(e,f))
FLOAT_THRESHOLD_EXPLICIT =
 |- threshold float_format = &340282356779733661637539395458142568448
VAL_THRESHOLD =
 |- ∀a. Finite a ⟹ abs(Val a) < threshold float_format
```

Preliminary theorems about rounding error

```
error =
 |- ∀x. error x = Val (float (round float_format To_nearest x)) - x
BOUND_AT_WORST_LEMMA =
 |- ∀a x.
        abs x < threshold float_format ∧ is_finite float_format a
        ⟹ abs(valof float_format (round float_format To_nearest x) - x)
            <= abs(valof float_format a - x)
ERROR_AT_WORST_LEMMA =
 |- ∀a x.
        abs x < threshold float_format ∧ Finite a
        ⟹ abs(error x) <= abs(Val a - x)
ERROR_IS_ZERO = |- ∀a x. Finite a ∧ (Val a = x) ⟹ (error x = &0)
```

Lemmas about general rounding error

```
ERROR_BOUND_LEMMA1 =
 |- ∀x. &0 <= x ∧ x < &1
        ⟹ (∃n. n < 2 EXP 23 ∧
                &n / &2 pow 23 <= x ∧
                x < &(SUC n) / &2 pow 23)
ERROR_BOUND_LEMMA2 =
 |- ∀x. &0 <= x ∧ x < &1
        ⟹ (∃n. n <= 2 EXP 23 ∧
                abs(x - &n / &2 pow 23) <= inv(&2 pow 24))
ERROR_BOUND_LEMMA3 =
 |- ∀x. &1 <= x ∧ x < &2
        ⟹ (∃n. n <= 2 EXP 23 ∧
                abs((&1 + &n / &2 pow 23) - x) <= inv(&2 pow 24))
ERROR_BOUND_LEMMA4 =
 |- ∀x. &1 <= x ∧ x < &2
        ⟹ (∃e f.
                abs(Val (float (0,(e,f))) - x) <= inv(&2 pow 24) ∧
                f < 2 EXP 23 ∧
                ((e = bias float_format) ∨
                 (e = SUC (bias float_format)) ∧ (f = 0)))
ERROR_BOUND_LEMMA5 =
 |- ∀x. &1 <= abs x ∧ abs x < &2
        ⟹ (∃s e f.
                abs(Val (float (s,(e,f))) - x) <= inv(&2 pow 24) ∧
                s < 2 ∧
                f < 2 EXP 23 ∧
                ((e = bias float_format) ∨
                 (e = SUC (bias float_format)) ∧ (f = 0)))
ERROR_BOUND_LEMMA6 =
 |- ∀x. &0 <= x ∧ x < inv(&2 pow 126)
        ⟹ (∃n. n <= 2 EXP 23 ∧
                abs(x - &2 / &2 pow 127 * &n / &2 pow 23) <=
                inv(&2 pow 150))
ERROR_BOUND_LEMMA7 =
 |- ∀x. &0 <= x ∧ x < inv(&2 pow 126)
        ⟹ (∃e f.
                abs(Val (float (0,(e,f))) - x) <= inv(&2 pow 150) ∧
                f < 2 EXP 23 ∧
                ((e = 0) ∨ (e = 1) ∧ (f = 0)))
ERROR_BOUND_LEMMA8 =
 |- ∀x. abs x < inv(&2 pow 126)
        ⟹ (∃s e f.
                abs(Val (float (s,(e,f))) - x) <= inv(&2 pow 150) ∧
                s < 2 ∧
                f < 2 EXP 23 ∧
                ((e = 0) ∨ (e = 1) ∧ (f = 0)))
```

General error bound theorems

```
VALOF_SCALE_UP =
 |- ∀s e k f.
        ¬(e = 0)
        ⟹ (valof float_format (s,(e + k,f)) =
             &2 pow k * valof float_format (s,(e,f)))
VALOF_SCALE_DOWN =
 |- ∀s e k f.
        k < e
        ⟹ (valof float_format (s,(e - k,f)) =
             inv(&2 pow k) * valof float_format (s,(e,f)))
ERROR_BOUND_BIG =
 |- ∀k x.
        &2 pow k <= abs x ∧
        abs x < &2 pow SUC k ∧
        abs x < threshold float_format
        ⟹ abs(error x) <= &2 pow k / &2 pow 24
ERROR_BOUND_SMALL =
 |- ∀k x.
        inv(&2 pow SUC k) <= abs x ∧ abs x < inv(&2 pow k) ∧ k < 126
        ⟹ abs(error x) <= inv(&2 pow SUC k * &2 pow 24)
ERROR_BOUND_TINY =
 |- ∀x. abs x < inv(&2 pow 126) ⟹ abs(error x) <= inv(&2 pow 150)
ERROR_BOUND_NORM_STRONG =
 |- ∀x j.
        abs x < threshold float_format ∧ abs x < &2 pow SUC j / &2 pow 126
        ⟹ abs(error x) <= &2 pow j / &2 pow 150
```

Rounding error in arithmetic operations

```
FLOAT_ADD =
 |- ∀a b.
        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))
FLOAT_SUB =
 |- ∀a b.
        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))
FLOAT_MUL =
 |- ∀a b.
        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))
FLOAT_DIV =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        ¬Iszero b ∧
        abs(Val a / Val b) < threshold float_format
        ⟹ Finite (a / b) ∧
            (Val (a / b) = Val a / Val b + error (Val a / Val b))
FLOAT_REM =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        ¬Iszero b ∧
        abs(Val a rem Val b) < threshold float_format
        ⟹ Finite (a % b) ∧
            (Val (a % b) = Val a rem Val b + error (Val a rem Val b))
FLOAT_SQRT =
 |- ∀a. Finite a ∧
        Val a >= &0 ∧
        abs(sqrt(Val a)) < threshold float_format
        ⟹ Finite (sqrt a) ∧
            (Val (sqrt a) = sqrt(Val a) + error (sqrt(Val a)))
FLOAT_NEG =
 |- ∀a. Finite a ⟹ Finite (--a) ∧ (Val (--a) = --(Val a))
FLOAT_ABS =
 |- ∀a. Finite a ⟹ Finite (abs a) ∧ (Val (abs a) = abs(Val a))
```

Perfect rounding if few significant digits

```
ERROR_ISZERO_SIGDIGITS_LEMMA1 =
 |- ∀p x e k.
        inv(&2 pow 126) <= x ∧
        x < threshold float_format ∧
        (x = &2 pow e / &2 pow 150 * &k) ∧
        p <= 24 ∧
        k < 2 EXP p
        ⟹ (∃a. Finite a ∧ (Val a = x))
ERROR_ISZERO_SIGDIGITS_LEMMA2 =
 |- ∀p x e k.
        inv(&2 pow 126) <= abs x ∧
        abs x < threshold float_format ∧
        (abs x = &2 pow e / &2 pow 150 * &k) ∧
        p <= 24 ∧
        k < 2 EXP p
        ⟹ (∃a. Finite a ∧ (Val a = x))
ERROR_ISZERO_SIGDIGITS_LEMMA3 =
 |- ∀p x e k.
        inv(&2 pow 126) <= abs x ∧
        abs x < threshold float_format ∧
        (abs x = &2 pow e / &2 pow 150 * &k) ∧
        k < 2 EXP 24
        ⟹ (∃a. Finite a ∧ (Val a = x))
ERROR_ISZERO_SIGDIGITS_LEMMA4 =
 |- ∀x e k.
        abs x < threshold float_format ∧
        (abs x = &2 pow e / &2 pow 149 * &k) ∧
        k < 2 EXP 24
        ⟹ (∃a. Finite a ∧ (Val a = x))
EXISTS_EXACT_SIGDIGITS =
 |- ∀x e k.
        (abs x = &2 pow e / &2 pow 149 * &k) ∧ k < 2 EXP 24 ∧ e < 254
        ⟹ (∃a. Finite a ∧ (Val a = x))
ERROR_ISZERO_SIGDIGITS =
 |- ∀x e k.
        (abs x = &2 pow e / &2 pow 149 * &k) ∧ k < 2 EXP 24 ∧ e < 254
        ⟹ (error x = &0)
ERROR_IS_ZERO_ADD =
 |- ∀a b c.
        Finite a ∧ Finite b ∧ Finite c ∧ (Val c = Val a + Val b)
        ⟹ Finite (a + b) ∧ (Val (a + b) = Val a + Val b)
ERROR_IS_ZERO_SUB =
 |- ∀a b c.
        Finite a ∧ Finite b ∧ Finite c ∧ (Val c = Val a - Val b)
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
ERROR_IS_ZERO_MUL =
 |- ∀a b c.
        Finite a ∧ Finite b ∧ Finite c ∧ (Val c = Val a * Val b)
        ⟹ Finite (a * b) ∧ (Val (a * b) = Val a * Val b)
```

Perfect rounding when subtraction cancels

```
FLOAT_SUB_CANCEL_LEMMA0 =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        (sign (defloat a) = sign (defloat b)) ∧
        (exponent (defloat a) = exponent (defloat b)) ∧
        fraction (defloat b) <= fraction (defloat a)
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
FLOAT_SUB_CANCEL_LEMMA1 =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        (sign (defloat a) = sign (defloat b)) ∧
        (exponent (defloat a) = exponent (defloat b))
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
FLOAT_SUB_CANCEL_LEMMA2a =
 |- ∀a b.
        Finite a ∧ Finite b ∧ abs(Val b) <= abs(Val a)
        ⟹ exponent (defloat b) <= exponent (defloat a)
FLOAT_SUB_CANCEL_LEMMA2 =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        abs(Val b) <= abs(Val a) ∧
        abs(Val a) <= &2 * abs(Val b)
        ⟹ (exponent (defloat a) = exponent (defloat b)) ∨
            (exponent (defloat a) = SUC (exponent (defloat b)))
FLOAT_SIGN_CASES =
 |- ∀a. (sign (defloat a) = 0) ∨ (sign (defloat a) = 1)
FLOAT_SIGN =
 |- ∀a. Finite a
        ⟹ (&0 < Val a ⟹ (sign (defloat a) = 0)) ∧
            (Val a < &0 ⟹ (sign (defloat a) = 1))
FLOAT_SIGN_01 =
 |- ∀a b. (sign (defloat a) = 1) ∧
          (sign (defloat b) = 0) ⟹ Val a <= Val b
FLOAT_SUB_CANCEL_LEMMA =
 |- ∀a b.
        Finite a ∧ Finite b ∧ Val b <= Val a ∧ Val a <= &2 * Val b
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
FLOAT_SUB_CANCEL =
 |- ∀a b.
        Finite a ∧ Finite b ∧ Val b / &2 <= Val a ∧ Val a <= &2 * Val b
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
FLOAT_SUB_CANCEL_ABS =
 |- ∀a b.
        Finite a ∧ Finite b ∧ &2 * abs(Val a - Val b) <= abs(Val a)
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
FLOAT_SUB_CANCEL_ABS2 =
 |- ∀a b.
        Finite a ∧ Finite b ∧ &2 * abs(Val a - Val b) <= abs(Val b)
        ⟹ Finite (a - b) ∧ (Val (a - b) = Val a - Val b)
```

Monotonicity of rounding

```
IS_CLOSEST_MONO =
 |- ∀v s x y a b.
        x < y ∧ is_closest v s x a ∧ is_closest v s y b ⟹ v a <= v b
CLOSEST_MONO =
 |- ∀v p s x y.
        FINITE s ∧ ¬(s = EMPTY) ∧ x <= y
        ⟹ v (closest v p s x) <= v (closest v p s y)
ROUND_MONO =
 |- ∀x y.
        x <= y ∧
        abs x < threshold float_format ∧
        abs y < threshold float_format
        ⟹ Val (float (round float_format To_nearest x)) <=
            Val (float (round float_format To_nearest y))
FLOAT_ADD_MONOROUND_LEMMA =
 |- ∀a b.
        Finite a ∧
        Finite b ∧
        abs(Val a + Val b) < threshold float_format ∧
        Val (a + b) < Val a
        ⟹ Val a + Val b < Val a
```

Relation between value and classification

```
ISNORMAL_VALUE =
 |- ∀a. Isnormal a = Finite a ∧ inv(&2 pow 126) <= abs(Val a)
ISZERO_VALUE = |- ∀a. Iszero a = Finite a ∧ (Val a = &0)
ISDENORMAL_VALUE =
 |- ∀a. Isdenormal a =
        Finite a ∧ abs(Val a) < inv(&2 pow 126) ∧ ¬(Val a = &0)
ISNORMAL_BININTERVAL_VALUE =
 |- ∀a. Isnormal a
        ⟹ &2 pow exponent (defloat a) / &2 pow 127 <= abs(Val a) ∧
            abs(Val a) < &2 pow SUC (exponent (defloat a)) / &2 pow 127
ISNORMAL_BININTERVAL_VALUE_STRONG =
 |- ∀a. Finite a
        ⟹ (∀p. 0 < p ∧
                &2 pow p / &2 pow 127 <= abs(Val a) ∧
                abs(Val a) < &2 pow SUC p / &2 pow 127 =
                Isnormal a ∧ (exponent (defloat a) = p))
FLOAT_BIG_ISINT =
 |- ∀a. Finite a ∧ exponent (defloat a) >= 150
        ⟹ is_integral float_format (defloat a)
```

Small integers are exactly representable

```
INT_SMALL_ISFLOAT_LEMMA1 =
 |- ∀p x.
        2 EXP p <= x ∧ x < 2 EXP SUC p ∧ p <= 23
        ⟹ is_finite float_format
            (0,(127 + p,(x - 2 EXP p) * 2 exp(23 - p))) ∧
            (Val (float (0,(127 + p,(x - 2 EXP p) * 2 exp(23 - p)))) = &x)
INT_SMALL_ISFLOAT_LEMMA2 =
 |- ∀p x.
        x < 2 EXP p ∧ p <= 24
        ⟹ (∃e f.
                 is_finite float_format (0,(e,f)) ∧
                 (Val (float (0,(e,f))) = &x))
INT_SMALL_ISFLOAT_LEMMA3 =
 |- ∀x. x <= 2 EXP 24
        ⟹ (∃e f.
                 is_finite float_format (0,(e,f)) ∧
                 (Val (float (0,(e,f))) = &x))
INT_SMALL_ISFLOAT =
 |- ∀x. abs x <= &2 pow 24 ∧ (∃n. abs x = &n)
        ⟹ (∃a. Finite a ∧ (Val a = x))
```

Rounding error when rounding to an integer

```
FLOAT_ROUNDFLOAT_LEMMA1 =
 |- ∀a. Finite a
        ⟹ is_closest (valof float_format)
                     {a | is_integral float_format a}
            (valof float_format (defloat a))
            (closest (valof float_format)
             (λa. ∃n. EVEN n ∧ (abs(valof float_format a) = &n))
             {a | is_integral float_format a}
            (valof float_format (defloat a)))
FLOAT_ROUNDFLOAT_LEMMA2 =
 |- ∀a. Finite a
        ⟹ (defloat
             (float
             (zerosign float_format (sign (defloat a))
             (closest (valof float_format)
              (λa. ∃n. EVEN n ∧ (abs(valof float_format a) = &n))
              {a | is_integral float_format a}
             (valof float_format (defloat a))))) =
             zerosign float_format (sign (defloat a))
             (closest (valof float_format)
              (λa. ∃n. EVEN n ∧ (abs(valof float_format a) = &n))
              {a | is_integral float_format a}
             (valof float_format (defloat a))))
FLOAT_ROUNDFLOAT_LEMMA3 =
 |- ∀b a.
        valof float_format (zerosign float_format b a) =
        valof float_format a
FLOAT_ROUNDFLOAT_LEMMA4 =
 |- ∀a. Finite a
        ⟹ (∃b. is_integral float_format b ∧
                abs(valof float_format b -
                     valof float_format (defloat a)) <= inv(&2))
FLOAT_ROUNDFLOAT =
 |- ∀a. Finite a
        ⟹ Isintegral (ROUNDFLOAT a) ∧
            abs(Val (ROUNDFLOAT a) - Val a) <= inv(&2)
ISINTEGRAL = |- ∀a. Isintegral a = Finite a ∧ (∃n. abs(Val a) = &n)
FLOAT_TOINT =
 |- ∀a. Isintegral a ∧ --(&2 pow 31) <= Val a ∧ Val a < &2 pow 31
        ⟹ (Ival (Toint a) = Val a)
FLOAT_TOFLOAT =
 |- ∀i. abs(Ival i) <= &2 pow 24
        ⟹ Finite (Tofloat i) ∧ (Val (Tofloat i) = Ival i)
```

Ordering of floating point values bitwise

```
FLOAT_ORDERED_OPPSIGN =
 |- ∀a b. sign (defloat a) < sign (defloat b) ⟹ Val b <= Val a
FLOAT_ORDERED_POS_LT =
 |- ∀a b.
        (sign (defloat a) = 0) ∧
        (sign (defloat b) = 0) ∧
        (exponent (defloat a) < exponent (defloat b) ∨
         (exponent (defloat a) = exponent (defloat b)) ∧
         fraction (defloat a) < fraction (defloat b))
        ⟹ Val a < Val b
FLOAT_ORDERED_POS_LE =
 |- ∀a b.
        (sign (defloat a) = 0) ∧
        (sign (defloat b) = 0) ∧
        (exponent (defloat a) < exponent (defloat b) ∨
         (exponent (defloat a) = exponent (defloat b)) ∧
         fraction (defloat a) <= fraction (defloat b))
        ⟹ Val a <= Val b
FLOAT_NEG_FIELDS =
 |- ∀a. Finite a
        ⟹ (Val
             (float
             (1 - sign (defloat a),
              (exponent (defloat a),fraction (defloat a)))) =
             --(Val a))
```

Placing of floats in binary intervals

```
FLOAT_BINARY_INTERVAL =
 |- ∀a. Isnormal a
        ⟹ &2 pow exponent (defloat a) / &2 pow 127 <= abs(Val a) ∧
            abs(Val a) < &2 * &2 pow exponent (defloat a) / &2 pow 127
FLOAT_BINARY_INTERVAL_UNIQUE =
 |- ∀a. Isnormal a ∧
        &2 pow n / &2 pow 127 <= abs(Val a) ∧
        abs(Val a) < &2 * &2 pow n / &2 pow 127
        ⟹ (exponent (defloat a) = n)
FLOAT_BINARY_INTERVAL_LE =
 |- ∀n a.
        Finite a ∧ abs(Val a) < &2 pow n / &2 pow 126
        ⟹ exponent (defloat a) <= n
FLOAT_ULP_POS = |- ∀a. &0 < Ulp a
FLOAT_ULP_EXPLICIT =
 |- ∀a. Ulp a =
        (if exponent (defloat a) = 0
         then inv(&2 pow 149)
         else &2 pow exponent (defloat a) / &2 pow 150)
FLOAT_ULP_NORMAL =
 |- ∀a. Isnormal a ⟹ (Ulp a = &2 pow exponent (defloat a) / &2 pow 150)
FLOAT_VAL_EXPONENT =
 |- ∀a b.
        Isnormal a ∧ Isnormal b ∧ (Val a = Val b)
        ⟹ (exponent (defloat a) = exponent (defloat b))
```

Error when scaling by 2^N (1)

```
FLOAT_SCALB =
 |- ∀a N.
        Finite a ∧
        abs(exp(Ival N * ln(&2)) * Val a) < threshold float_format
        ⟹ Finite (Scalb (a,N)) ∧
            (Val (Scalb (a,N)) =
             exp(Ival N * ln(&2)) * Val a +
             error (exp(Ival N * ln(&2)) * Val a))
FLOAT_SCALB_OVERFLOWS_SIGNED =
 |- ∀a N.
        Finite a
        ⟹ (exp(Ival N * ln(&2)) * Val a >= threshold float_format
             ⟹ Scalb (a,N) == Plus_infinity) ∧
            (exp(Ival N * ln(&2)) * Val a <= --(threshold float_format)
             ⟹ Scalb (a,N) == Minus_infinity)
FLOAT_SCALB_OVERFLOWS =
 |- ∀a N.
        Finite a ∧
        abs(exp(Ival N * ln(&2)) * Val a) >= threshold float_format
        ⟹ Infinity (Scalb (a,N))
FLOAT_SCALE_UP =
 |- ∀n a.
        Isnormal a ∧ abs(&2 pow n * Val a) < threshold float_format
        ⟹ (∃b. Isnormal b ∧
                (exponent (defloat b) = exponent (defloat a) + n) ∧
                (Val b = &2 pow n * Val a))
FLOAT_SCALE_DOWN_NORM =
 |- ∀n a.
        Isnormal a ∧ inv(&2 pow 126) <= abs(Val a / &2 pow n)
        ⟹ (∃b. Isnormal b ∧
                (exponent (defloat a) = exponent (defloat b) + n) ∧
                (Val b = Val a / &2 pow n))
REAL_EXP_IVAL_LEMMA =
 |- ∀N. (∃n. exp(Ival N * ln(&2)) = &2 pow n) ∨
        (∃n. exp(Ival N * ln(&2)) = inv(&2 pow n))
```

Error when scaling by 2^N (2)

```
FLOAT_SCALB_NORM =
 |- ∀a N.
        Isnormal a ∧
        inv(&2 pow 126) <= abs(exp(Ival N * ln(&2)) * Val a) ∧
        abs(exp(Ival N * ln(&2)) * Val a) < threshold float_format
        ⟹ Finite (Scalb (a,N)) ∧
            Isnormal (Scalb (a,N)) ∧
            (Val (Scalb (a,N)) = exp(Ival N * ln(&2)) * Val a) ∧
            (Ulp (Scalb (a,N)) = exp(Ival N * ln(&2)) * Ulp a)
FLOAT_NORM_MINIMUM =
 |- ∀a. Finite a ∧ abs(Val a) > inv(&2 pow 126)
        ⟹ abs(Val a) >= inv(&2 pow 126) + inv(&2 pow 150)
FLOAT_SCALB_DENORM =
 |- ∀a N.
        Isnormal a ∧ abs(exp(Ival N * ln(&2)) * Val a) < inv(&2 pow 126)
        ⟹ Finite (Scalb (a,N)) ∧
            abs(Val (Scalb (a,N))) <= inv(&2 pow 126) ∧
            abs(Val (Scalb (a,N)) - exp(Ival N * ln(&2)) * Val a) <=
            inv(&2) * Ulp (Scalb (a,N)) ∧
            (Ulp (Scalb (a,N)) = inv(&2 pow 149)) ∧
            Ulp (Scalb (a,N)) >= &2 * exp(Ival N * ln(&2)) * Ulp a
```

Abbreviating definition for table entries

```
TABLES_OK =
 |- 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 2) = float (0,(127,371392))) ∧
    (S_Lead (Int 3) = float (0,(127,563200))) ∧
    (S_Lead (Int 4) = float (0,(127,759232))) ∧
    (S_Lead (Int 5) = float (0,(127,959488))) ∧
    (S_Lead (Int 6) = float (0,(127,1164224))) ∧
    (S_Lead (Int 7) = float (0,(127,1373376))) ∧
    (S_Lead (Int 8) = float (0,(127,1587136))) ∧
    (S_Lead (Int 9) = float (0,(127,1805568))) ∧
    (S_Lead (Int 10) = float (0,(127,2028800))) ∧
    (S_Lead (Int 11) = float (0,(127,2256960))) ∧
    (S_Lead (Int 12) = float (0,(127,2490048))) ∧
    (S_Lead (Int 13) = float (0,(127,2728256))) ∧
    (S_Lead (Int 14) = float (0,(127,2971648))) ∧
    (S_Lead (Int 15) = float (0,(127,3220416))) ∧
    (S_Lead (Int 16) = float (0,(127,3474624))) ∧
    (S_Lead (Int 17) = float (0,(127,3734400))) ∧
    (S_Lead (Int 18) = float (0,(127,3999872))) ∧
    (S_Lead (Int 19) = float (0,(127,4271168))) ∧
    (S_Lead (Int 20) = float (0,(127,4548352))) ∧
    (S_Lead (Int 21) = float (0,(127,4831616))) ∧
    (S_Lead (Int 22) = float (0,(127,5121152))) ∧
    (S_Lead (Int 23) = float (0,(127,5416960))) ∧
    (S_Lead (Int 24) = float (0,(127,5719232))) ∧
    (S_Lead (Int 25) = float (0,(127,6028160))) ∧
    (S_Lead (Int 26) = float (0,(127,6343872))) ∧
    (S_Lead (Int 27) = float (0,(127,6666496))) ∧
    (S_Lead (Int 28) = float (0,(127,6996160))) ∧
    (S_Lead (Int 29) = float (0,(127,7332992))) ∧
    (S_Lead (Int 30) = float (0,(127,7677248))) ∧
    (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 2) = float (0,(105,5894930))) ∧
    (S_Trail (Int 3) = float (0,(107,6818094))) ∧
    (S_Trail (Int 4) = float (0,(104,7468358))) ∧
    (S_Trail (Int 5) = float (0,(109,6696215))) ∧
    (S_Trail (Int 6) = float (0,(108,1809753))) ∧
    (S_Trail (Int 7) = float (0,(109,4105212))) ∧
    (S_Trail (Int 8) = float (0,(109,4277815))) ∧
    (S_Trail (Int 9) = float (0,(109,6743893))) ∧
    (S_Trail (Int 10) = float (0,(109,4817479))) ∧
    (S_Trail (Int 11) = float (0,(105,4244242))) ∧
    (S_Trail (Int 12) = float (0,(108,3493259))) ∧
    (S_Trail (Int 13) = float (0,(108,5613396))) ∧
    (S_Trail (Int 14) = float (0,(109,8104203))) ∧
    (S_Trail (Int 15) = float (0,(109,5637963))) ∧
    (S_Trail (Int 16) = float (0,(109,5033959))) ∧
    (S_Trail (Int 17) = float (0,(109,4005260))) ∧
    (S_Trail (Int 18) = float (0,(109,949600))) ∧
    (S_Trail (Int 19) = float (0,(107,5023335))) ∧
    (S_Trail (Int 20) = float (0,(109,2639188))) ∧
    (S_Trail (Int 21) = float (0,(109,7739833))) ∧
    (S_Trail (Int 22) = float (0,(107,4280824))) ∧
    (S_Trail (Int 23) = float (0,(108,7114633))) ∧
    (S_Trail (Int 24) = float (0,(109,7547738))) ∧
    (S_Trail (Int 25) = float (0,(109,6184812))) ∧
    (S_Trail (Int 26) = float (0,(108,7823701))) ∧
    (S_Trail (Int 27) = float (0,(106,6090640))) ∧
    (S_Trail (Int 28) = float (0,(106,6093733))) ∧
    (S_Trail (Int 29) = float (0,(109,6713203))) ∧
    (S_Trail (Int 30) = float (0,(109,7623826))) ∧
    (S_Trail (Int 31) = float (0,(109,4943305)))
```

Explicit approximations to $ln(2)$

```
REAL_EXP_LINAPPROX_LEMMA =
 |- ∀x. abs x < Val (float (0,(102,0)))
        ⟹ abs(exp x - (&1 + x)) < &4 / &100 * inv(&2 pow 24)
LN_2_COMPOSITION = |- ln(&2) = ln(&1 + inv(&2)) - ln(&1 - inv(&4))
LN_12_APPROX =
 |- ∀n. 0 < n
        ⟹ abs
            (ln(&1 + inv(&2)) -
             Sum (0,n) (λm. --(&1) pow SUC m * inv(&m * &2 pow m))) <=
            inv(&n * &2 pow n)
LN_34_APPROX =
 |- ∀n. 0 < n
        ⟹ abs(ln(&1 - inv(&4)) + Sum (0,n) (λm. inv(&m * &4 pow m))) <=
            inv(&n * &3 pow n)
LN_2_APPROX =
 |- abs
    (ln(&2) -
     (Sum (0,47) (λm. --(&1) pow SUC m * inv(&m * &2 pow m)) +
      Sum (0,30) (λm. inv(&m * &4 pow m)))) <
    inv(&2 pow 51)
LN_2_NUMERIC =
 |- abs
    (ln(&2) -
     &5445319802026545833408256866208 47 /
     &78559358744381708183222972579840 0) <
    inv(&2 pow 51)
```

Special case of cancellation

```
AD_HOC_L1_LEMMA1 =
 |- ∀X. Finite X ∧
        Val (float (0,(120,0))) <= Val X ∧
        Val X <= Val (float (0,(121,3240448)))
        ⟹ Finite (float (0,(121,3240448)) - X) ∧
            (Val (float (0,(121,3240448)) - X) =
             Val (float (0,(121,3240448))) - Val X)
AD_HOC_L1_LEMMA2a =
 |- ∀X L1.
        (L1 = float (0,(121,3240448))) ∧
        Finite X ∧
        abs(Val X - Val L1) <= inv(&88)
        ⟹ Finite (L1 - X) ∧ (Val (L1 - X) = Val L1 - Val X)
AD_HOC_L1_LEMMA2b =
 |- ∀X L1.
        (L1 = float (0,(121,3240448))) ∧
        Finite X ∧
        abs(Val X - Val L1) <= inv(&88)
        ⟹ Finite (X - L1) ∧ (Val (X - L1) = Val X - Val L1)
AD_HOC_L1_LEMMA2c =
 |- ∀X L1.
        (L1 = float (0,(121,3240448))) ∧
        Finite X ∧
        abs(Val X + Val L1) <= inv(&88)
        ⟹ Finite (X + L1) ∧ (Val (X + L1) = Val X + Val L1)
AD_HOC_L1_LEMMA3 =
 |- ∀X N L1.
        (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))
REAL_SUB_ISINTEGER_NUM = |- ∀m n. ∃k. abs(&m - &n) = &k
REAL_SUB_ISINTEGER_LEMMA =
 |- ∀a b. (∃n. abs a = &n) ∧ (∃n. abs b = &n) ⟹ (∃n. abs(a - b) = &n)
```

Approximations to $2^{\frac{j}{32}}$

```
ROOT_2_LEMMA0a = |- ∀x. ((λx. x pow 32) diffl &32 * x pow 31) x
ROOT_2_LEMMA0b =
 |- ∀x. (λx. x pow 32) differentiable x ∧ (λx. x pow 32) contl x
ROOT_2_LEMMA1 =
 |- ∀a b.
        a <= b
        ⟹ (∃x. a <= x ∧
                x <= b ∧
                (b pow 32 = a pow 32 + &32 * x pow 31 * (b - a)))
ROOT_32_POW = |- ∀x. &0 <= x ⟹ (root 32 x pow 32 = x)
ROOT_32_POWJ_POW = |- ∀j. root 32 (&2 pow j) pow 32 = &2 pow j
ROOT_2_LEMMA2a =
 |- ∀j x.
        &0 <= x ∧ x pow 32 >= &2 pow j
        ⟹ x >= root 32 (&2 pow j) ∧
            x pow 32 - &2 pow j >=
            &32 * root 32 (&2 pow j) pow 31 * (x - root 32 (&2 pow j))
ROOT_2_LEMMA2b =
 |- ∀j x.
        &0 <= x ∧ x pow 32 <= &2 pow j
        ⟹ x <= root 32 (&2 pow j) ∧
            &2 pow j - x pow 32 >=
            &32 * x pow 31 * (root 32 (&2 pow j) - x)
ROOT_2_LEMMA3a =
 |- ∀j x.
        &0 < x ∧ x <= &2 ∧ x pow 32 >= &2 pow j
        ⟹ abs(x - root 32 (&2 pow j)) <=
            abs(x pow 32 - &2 pow j) / (&16 * &2 pow j)
ROOT_2_LEMMA3b =
 |- ∀j x.
        &0 < x ∧ x pow 32 <= &2 pow j
        ⟹ abs(x - root 32 (&2 pow j)) <=
            abs(x pow 32 - &2 pow j) / (&32 * x pow 32 / x)
ROOT_2_APPROX =
 |- ∀j x.
        &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)
```

Properties of tables

```
TABLES_OK_APPROX =
 |- ∀S_Lead S_Trail.
        TABLES_OK S_Lead S_Trail
        ⟹ (∀j. 0 <= j ∧ j < 32
                ⟹ abs
                    ((Val (S_Lead (Int j)) + Val (S_Trail (Int j))) -
                     root 32 (&2 pow j)) <=
                    inv(&2 pow 41))
TABLES_OK_FINITE =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32
            ⟹ Finite (S_Lead (Int i)) ∧
                Finite (S_Trail (Int i)) ∧
                abs(Val (S_Lead (Int i)) + Val (S_Trail (Int i))) <
                threshold float_format)
TABLES_OK_LT =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32
            ⟹ abs(Val (S_Lead (Int i)) + Val (S_Trail (Int i))) <=
                &4303805443529 / &2199023255552)
TABLES_OK_LT_2_STRONG =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32
            ⟹ abs(Val (S_Lead (Int i)) + Val (S_Trail (Int i))) <
                &2 - inv(&2 pow 24))
TABLES_OK_LT_2 =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32
            ⟹ abs(Val (S_Lead (Int i)) + Val (S_Trail (Int i))) < &2)
TABLES_OK_GT =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. ¬(i = 0) ∧ i < 32
            ⟹ &17977404757381 / &17592186044416 <=
                Val (S_Lead (Int i)) + Val (S_Trail (Int i)))
TABLES_OK_LEAD_GT =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32 ⟹ &1 <= Val (S_Lead (Int i)))
TABLES_OK_GT_1 =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32
            ⟹ &1 <= Val (S_Lead (Int i)) + Val (S_Trail (Int i)))
TABLES_OK_TRAIL_LT =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32 ⟹ abs(Val (S_Trail (Int i))) <= inv(&133332))
TABLES_OK_LEAD_LT =
 |- TABLES_OK S_Lead S_Trail
    ⟹ (∀i. i < 32 ⟹ abs(Val (S_Lead (Int i))) <= &128263 / &65536)
```

Analysis of overflow

```
LN_THRESHOLD_NUMERIC =
 |- abs
    (ln(threshold float_format) -
     &5445319800197443227259292415904 47 /
     &61374499019048209518142947328 00) < inv(&2 pow 43)
LN_THRESHOLD_STRADDLE =
 |- Val (float (0,(133,3240471))) < ln(threshold float_format) ∧
    ln(threshold float_format) < Val (float (0,(133,3240472)))
LN_THRESHOLD_AWAY_CLOSEST =
 |- abs(ln(threshold float_format) - Val (float (0,(133,3240471)))) >
    inv(&2 pow 22) ∧
    abs(ln(threshold float_format) - Val (float (0,(133,3240472)))) >
    inv(&2 pow 22)
LN_THRESHOLD_AWAY_LEMMA =
 |- ∀x. Finite x
        ⟹ Val x <= Val (float (0,(133,3240471))) ∨
           Val x >= Val (float (0,(133,3240472)))
LN_THRESHOLD_AWAY =
 |- ∀a. Finite a
        ⟹ abs(ln(threshold float_format) - Val a) > inv(&2 pow 22)
```

Analysis of underflow

```
LN_UNDERTHRESH_NUMERIC =
 |- abs
    (ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) -
     --(&490078782516853601559696617442922 3) /
     &561138276745583629880164089856 00) <
    &67 / &128 * inv(&2 pow 43)
LN_UNDERTHRESH_STRADDLE =
 |- Val (float (1,(133,3058768))) <
    ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) ∧
    ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) <
    Val (float (1,(133,3058767)))
LN_UNDERTHRESH_AWAY_CLOSEST =
 |- abs
    (ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) -
     Val (float (1,(133,3058768)))) >
    &11 / &10 * inv(&2 pow 19) ∧
    abs
    (ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) -
     Val (float (1,(133,3058767)))) >
    &11 / &10 * inv(&2 pow 19)
LN_UNDERTHRESH_AWAY_POS_LEMMA =
 |- ∀x. Finite x
        ⟹ Val x <= Val (float (0,(133,3058767))) ∨
           Val (float (0,(133,3058768))) <= Val x
LN_UNDERTHRESH_AWAY_LEMMA =
 |- ∀x. Finite x
        ⟹ Val x <= Val (float (1,(133,3058768))) ∨
           Val x >= Val (float (1,(133,3058767)))
LN_UNDERTHRESH_AWAY =
 |- ∀a. Finite a
        ⟹ abs(ln(inv(&2 pow 126) * (&1 - inv(&2 pow 24))) - Val a) >
           &11 / &10 * inv(&2 pow 19)
EXP_UNDERTHRESH_AWAY =
 |- ∀X. Finite X
        ⟹ ¬(abs
              (exp(Val X) / (inv(&2 pow 126) * (&1 - inv(&2 pow 24))) -
               &1) <=
              inv(&2 pow 19))
```

Final correctness theorem

```
|- (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))
```

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.

Beeson, M. J. (1984) *Foundations of constructive mathematics: metamathematical studies*, Volume 3 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag.

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

Boulton, R. et al. (1993) Experience with embedding hardware description languages in HOL. In Stavridou, V., Melham, T. F., and Boute, R. T. (eds.), *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, Volume A-10 of *IFIP Transactions A: Computer Science and Technology*, Nijmegen, The Netherlands, pp. 129–156. North-Holland.

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

Church, A. (1940) A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**, 56–68.

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.

Grundy, J. (1993) Predicative programming—a survey. In Bjørner, D., Broy, M., and Pottosin, I. V. (eds.), *Formal Methods in Programming and Their Applications: Proceedings of the International Conference*, Volume 735 of *Lecture Notes in Computer Science*, Academgorodok, Novosibirsk, Russia, pp. 8–25. 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.

Henkin, L. (1963) A theory of propositional types. *Fundamenta Mathematicae*, **52**, 323–344.

Hesselink, W. H. (1992) *Programs, Recursion and Unbounded Choice*, Volume 27 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

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.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**, 348–375.

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. (1996) How to believe a machine-checked proof. Submitted to the 1997 Venice type theory meeting; currently available on the Web as `http://www.brics.dk/~pollack/export/BelieveMachineCheckedProof.ps.gz`.

Pour-El, M. B. and Richards, J. I. (1980) *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer-Verlag.

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.

Remes, M. E. (1934) Sur le calcul effectif des polynomes d'approximation de Tchebichef. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, **199**, 337–340.

Rivlin, T. J. (1962) Polynomials of best uniform approximation to certain rational functions. *Numerische Mathematik*, **4**, 345–349.

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.

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.

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.