Under consideration for publication in Formal Aspects of Computing

Formal Verification of Transcendental Fixed and Floating Point Algorithms using an Automatic Theorem Prover

Samuel Coward[1], Lawrence Paulson[2], Theo Drane[3] and Emiliano Morini[3]

[1]Faculty of Mathematics, University of Cambridge,
[2]Computer Laboratory, University of Cambridge,
[3]Cadence Design Systems, Cambridge

Abstract. We present a method for formal verification of transcendental hardware and software algorithms that scales to higher precision without suffering an exponential growth in runtimes. A class of implementations using piecewise polynomial approximation to compute the result is verified using MetiTarski, an automated theorem prover, which verifies a range of inputs for each call. The method was applied to commercial implementations from Cadence Design Systems with significant runtime gains over exhaustive testing methods and was successful in proving that the expected accuracy of one implementation was overly optimistic. Reproducing the verification of a sine implementation in software, previously done using an alternative theorem proving technique, demonstrates that the MetiTarski approach is a viable competitor. Verification of a 52 bit implementation of the square root function highlights the method's high precision capabilities.

Keywords: Theorem prover; Transcendental functions; Floating point algorithms; Hardware implementation

Declarations

Funding: The conceptualisation and much of the preliminary work for this paper was done during a summer internship at Cadence Design Systems. The development of MetiTarski was supported by EPSRC grants EP/C013409/1 and EP/I011005/1.
Conflicts of interest: Samuel Coward, Theo Drane and Emiliano Morini have now moved on to new roles and are no longer affiliated with Cadence Design Systems.
Availability of data and material: All problem scripts, excluding the examples verifying commercial

Correspondence and offprint requests to: Samuel Coward, 58 Grove Hall Court, 2 Hall Road, St Johns Wood, London, NW8 9NY, UK.
e-mail: s.coward111@gmail.com

Cadence Design Systems' implementations, can be found on GitHub
`https://github.com/SRCoward/verification_2018.git`.
Code availability: MetiTarski is an open source theorem prover. It can be downloaded from
`https://www.cl.cam.ac.uk/~lp15/papers/Arith/`. It must be paired with an additional solver, for which there are several options documented in the MetiTarski user guide.
Authors' contributions: The study conception and design was led by Theo Drane and Emiliano Morini. Material preparation and analysis were performed by Samuel Coward, Lawrence Paulson and Emiliano Morini. The first draft of the manuscript was written by Samuel Coward and edited by Lawrence Paulson.

1. Introduction

Formal verification of floating point operations is becoming ever more challenging as hardware designs reach levels of complexity only previously seen in software. Its importance in industry is well known, exemplified by the Pentium floating point division bug [Pra95]. We present a new approach to the verification of fixed and floating point transcendental algorithms. This technique should be viable for verifying high precision algorithms, as our findings suggest that runtimes will not rise exponentially with the precision. Our experiments cover implementations of logarithms, square root and the sine function; however, the methodology can also be applied to many different functions implemented using algorithms of the form described in §2. Verification of logarithm implementations is a relevant problem as it finds applications in fields such as digital signal processing and 3D graphics [Lew95, Har01]. In addition, it can be very simple to implement, with one of the simplest examples of a floating point logarithm using the exponent as the integer part of the result combined with a lookup table (LUT) for the most significant bits (msb) of the significand, to generate the fractional part [Har01].

$$\log(2^{exp} \times 1.sig) \approx exp + LUT(sig[msb])$$

Traditional techniques rely on exhaustive testing of all inputs to verify such algorithms, but this can be resource intensive, perhaps prohibitively so. The Multiple Precision Floating-Point Reliable (MPFR) library is a C library for multiple-precision floating point computations with correct rounding [FHL+07], and is widely used as a reference for many verification tasks. For example, some of the industrial implementations presented here were verified by comparing the outputs to the MPFR library. We shall see that the methodology used in this paper performs more efficiently in particular cases.

The paper will focus on implementations of transcendental functions in hardware that rely on piecewise polynomial approximations. Many elementary functions are traditionally calculated in software [Cod80, Gal91], but for numerically intensive algorithms such implementations may simply be too slow, leading to the development of dedicated hardware [Tan91, SDCP11, PEB04]. Although primarily focusing on hardware, some software implementations may be amenable to the verification approach presented here, as we shall see in §6. De Dinechin, Lauter and Melquiond verified a CRlibm binary64 library function using the Gappa proof assistant [LDDD+09, dLM11]. Their approach is potentially the most similar method (in execution) to ours and therefore we will also use our method to verify the same function. All the examples considered here use a binary representation, but the simplest decimal floating point implementations, that convert to binary, use the binary algorithm, then convert back to decimal would also be amenable [Har09]. Other decimal floating point implementations appear to rely more on the digit-recurrence algorithm [CZC+09, CHCK12], which is more challenging to reduce to a series of inequalities since decisions are typically made at each iteration based on information from the previous iterations. Reducing the problem to a series of inequalities is important as this is a form which the chosen theorem prover can solve.

To produce the required proofs we use MetiTarski [AP10], an automatic theorem prover for real valued analytic functions, such as cosine and logarithm. It's a combination of a resolution theorem prover and a decision procedure for the theory of real closed fields (RCF). An ordered field is real closed if every positive number has a square root and every odd-degree polynomial has at least one root, which is equivalent to saying that the field has the same first-order properties as the reals. The resolution prover at the base of MetiTarski is Joe Hurd's Metis [LH07], which is modified in several respects [AP08]. One key modification is to the ordering of the resolution prover [LW07], which encourages the replacement of supported functions by bounds. The inbuilt axioms are primarily upper and lower bounds on a set of supported functions. The choice of these bounds was carefully considered. Many are based on the bounds proposed by Daumas, Lester and Munoz [DLM08]; these are typically derived from Taylor series. Other bounds are obtained from continued

fraction expansions, for example

$$\frac{1}{2} \leq x \leq 1 \Rightarrow \frac{x-1}{x} \leq \ln(x) \leq \frac{3x^2-4x+1}{2x^2}.$$

The resolution prover applies the axioms, replacing any supported funtions and generates polynomial inequalites. With the problem reduced to the theory of RCFs, the decision procedure is then sufficient to finalise the proof. Conjectures are passed to MetiTarski as a set of inequalities which are transformed by replacing any special function by an appropriate bound. Typically proofs are found in a few seconds [AP09], but if MetiTarski is unable to prove a conjecture it does not mean that the conjecture is false. For verification it is important that MetiTarski produces machine readable proofs that include algebraic simplification, decision procedure calls, and resolution rules [DAT+09]. MetiTarski is among a limited number of automated tools supporting transcendental functions. Examples include the first-order logic automated reasoning tool `dReal` [GKC13] and an approach to Satisfiability Modulo the theory of transcendental functions, which used properties from the MetiTarski suite as benchmarks [CGI+17]. Our initial research encouraged the MetiTarski developers to add an axiom to bound the floor function. The main benefit of the update, for this paper, is to simplify the syntax and construction of our conjectures.

The implementations we study will use a piecewise polynomial approach. To approximate a function $f(x)$, for $x \in \Sigma$, where Σ is some input domain, we take a set of polynomials $p_i(x)$ for $i = 0, 1..., K$ and $x \in \Sigma_i$, where $\Sigma = \bigcup_i \Sigma_i$ and $\Sigma_i \cap \Sigma_j = \emptyset, \forall i \neq j$. The piecewise polynomial implementation is generally accompanied by some claimed error bound, ϵ, which is what we aim to prove. More precisely, we prove that

$$\forall i = 0, 1, ..., K \text{ and } \forall x \in \Sigma_i, \ |f(x) - p_i(x)| < \epsilon.$$

For each i, we will generate at least one problem to be automatically proven by MetiTarski. The main novelty of this contribution is in the application of MetiTarski to this verification problem. MetiTarski's understanding of elementary functions means that it is the only necessary theorem proving tool required in this methodology, unlike other approaches that have combined tools [dLM11]. The automatic generation of problem files from a template problem makes the verification effort simpler and could be used with other automatic theorem provers.

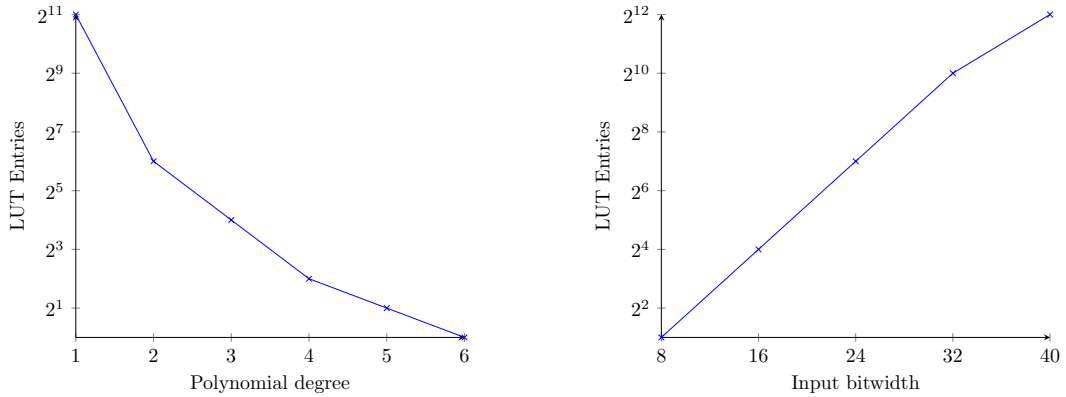
In §2 we will discuss the common approaches to implementations of elementary functions. In §3 we will describe the verification methodology of this paper, with actual applications and results of using this method presented in §4 and §5. Lastly, we will compare our approach to other theorem proving verification methods in §6 and §7.

2. Transcendental Function Implementations

As mentioned above, transcendental functions are commonly implemented in software, however the numerical algorithms used in software are often unsuitable for hardware. One example, using Chebyshev series expansion, would result in a high area circuit due to its use of a variety of expensive operations such as floating point multiplication [Fow93]. There are lots of different hardware algorithms to implement these functions, but a large proportion fall into one of the following categories: digit-recurrence [BKM94, PEB02], CORDIC (COordinate Rotation DIgital Computer) [Vol59, And98, Wal71] or table-lookup [Tan91, ST99]. A comparison of the different algorithms is beyond the scope of this paper but has been tackled by other authors [Tan91, PEB04]. We will focus on table-lookup algorithms, as they are broadly used and are the most amenable to our methodology .

Tang's 1991 paper provided a general framework for implementing a function f on an interval I, which most table driven algorithms use to some degree [Tan91]. According to Tang, typical table-lookup algorithms have a set of *breakpoints* $\{c_1, ..., c_N\}$, where $c_k \in I$ for $k = 1, 2, ...N$, along with a table of approximations T, such that $f(c_k) \approx T_k$, for $T_k \in T$. Given $x \in I$, the algorithm uses the following steps to calculate $f(x)$:

1. **Reduction**: Solve $k = min_k|x - c_k|$, then apply a *reduction transformation* $r = R(x, c_k)$.
2. **Approximation**: Approximate $f(r)$ using a function $p(r)$; often a polynomial is used here.
3. **Reconstruction**: Using a reconstruction function S, which is determined by f and R, find a final



(a) Exponential relationship between polynomial degree and LUT entries.

(b) Using quadratic interpolation for increasing input bitwidths.

Fig. 1. FloPoCo [dDP11] generated piecewise polynomial approximations to e^x, for $x \in [0, 1]$ and a 1 ULP (unit in last place) error bound.

approximation.

$$\begin{aligned} f(x) &= S(f(c_k), f(r)) \\ &\approx S(T_k, p(r)). \end{aligned}$$

In the rest of the paper, Tang describes algorithms for 2^x, $\log(x)$, and $\sin(x)$, for relatively narrow intervals. In these examples, tables ranging in size from 32 to 64 entries are used. To support wider intervals, further transformations of the arguments to these narrow domains are necessary. For example, Tang proposes an algorithm to compute $\ln(x)$ for $x \in [1, 2]$, which uses breakpoints $c_k = 1 + k/64$ for $k = 0, 1, ..., 64$. The breakpoint is chosen which satisfies $|x - c_k| < 1/128$ and a reduced argument $r = 2(x - c_k)/(x + c_k)$ is used to compute a polynomial approximation $p(r)$. The final approximation is given by $\ln(x) \approx T_k + p(r)$, where $T_k \approx \ln(c_k)$ are the tabulated values [Tan91]. Polynomials are a common choice for approximating with a reduced argument, and to calculate the coefficients there are a number of approaches. Some use the Remez algorithm to generate the coefficients of the minmax polynomial [Vei60, Tan91], while others opt to use carefully rounded coefficients from the function's Chebyshev expansion [SS93]. For the IA-64 architecture, Intel provided a library of table based algorithms for computing several transcendental functions [HKS+99]. The tables used in this library range in size from 24 to 256 double extended entries, for the exponential and logarithm, respectively.

Table based algorithms have been further developed and modified to use lookup tables (LUTs) to construct piecewise polynomial approximations to some elementary functions [SDCP11, POMB05]. The reduction step still uses the breakpoint method, but the table no longer returns just an approximation, T_k, it now returns multiple polynomial coefficients for each lookup. Strollo, De Caro and Petra present, alongside their algorithm, many different divisions of the input domain. For example, to compute $\ln(1 + x)$ for $x \in [0, 1]$ using a piecewise-quadratic approximation, accurate to 24 fractional bits, they required 128 table entries of coefficients [SDCP11]. In the industrial implementation described below, a table containing 32 breakpoints is used.

To further understand the relationship between LUT size and architecture choices, we can experiment with the FloPoCo tool [dDP11]. FloPoCo can automatically generate piecewise polynomial approximations to any supported function that are accurate to 1 ULP (unit in last place) for a given bitwidth. Note that in such tools, changing the function being approximated only changes the polynomials and doesn't fundamentally change the architecture. Therefore, the architecture is to some degree independent of the function it approximates. Figure 1 shows how the LUT size required changes with the choice of polynomial degree. The second graph highlights how if we use quadratic polynomials and increase the precision of the approximation, exponentially more LUT entries are required. FloPoCo will be looked at in greater detail in §5.

In this paper, the input to the function will often be represented in floating point format [Gol91], which stores a number as $(-1)^s \times 2^{e-b} \times 1.\mathit{significand}$. In IEEE-754 standard single precision, s is a single bit

representing the sign, the exponent e is an 8 bit integer, the bias b is a constant equal to 127, and the significand is 23 bits long. The implementations of the logarithm verified in this paper rely on the following identity:

$$\log(2^{exp} \times 1.significand) = \log(2^{exp}) + \log(1.significand) \quad (1)$$
$$= exp + \log(1.significand). \quad (2)$$

The reconstruction step then simply involves adding the integer exponent to an approximation to $\log(1.significand)$. This approximation passes the top k bits of the significand to a lookup table, which returns coefficients for a degree m polynomial, evaluated using the remaining low bits of the significand. Clearly, if the polynomial is a constant polynomial, then that is equivalent to the T_k described above. As this approach is essentially an enhancement to the method described by Tang, the verification of these piecewise polynomial LUT methods could also be adapted to the simpler LUT methods.

3. Verification Methodology

Given an implementation of a transcendental function following the outline above, we obtain an abstraction that is verifiable using MetiTarski. For a single precision implementation, if the top k bits of the significand are passed to a lookup table, for a fixed 8-bit exponent and sign bit, we reduce the verification problem to 2^k calls to MetiTarski. Therefore, the full verification over all inputs is reduced to just 2^{k+8+1} MetiTarski conjectures to be proven. In some cases, verification over all such inputs is not necessary: a bespoke hand proof may be able to confirm the correctness of the results for exponent scalings. Of course, most verification tasks use massively parallel processing to reduce the runtimes. Similar methods may be used to reduce the runtimes in our approach, as the conjectures passed to MetiTarski are independent of each other. In nearly all commercial implementations, k is relatively small as lookup tables can have high ROM demands [SDCP11]. Assuming that our interpolation coefficients are stored in a file, and we can express the problem as a set of inequalities, the procedure follows the same basic outline.

Procedure Outline (see Figure 2)

1. Write a template problem for the inequalities to be proven, with placeholders for all the relevant interpolation coefficients and most significant bit values.
2. Use a wrapper script to read the coefficients and replace them in the template problem to generate the full set of MetiTarski problems.
3. Run the Perl script that accompanies MetiTarski to test all of the problems.
4. Refine error modeling on problems that are not proven and return to step 1.
5. Exhaustively test regions where MetiTarski was unsuccessful.

To demonstrate the methodology, we analyse a toy implementation for computing the natural logarithm based on the outline above. The implementation takes as an input an 8 bit integer $x_0x_1...x_7$ and outputs an approximation to $\ln(1.x_0...x_7)$. The top four bits, $i = x_0..x_3$, are passed to a lookup table that returns the 10 bit interpolation coefficients a_i, b_i, c_i, for $i = 0, ..., 15$. The coefficients are generated using a simple quadratic interpolation scheme. Writing $x = 0.x_0...x_7$, the approximation generated is,

$$\overline{\ln(1+x)} = c_i + b_i x + a_i x^2.$$

This example is designed to be simple to show the underlying principles of the verification methodology. Later, we shall adapt it to be more relevant to industrial implementations. In this case, the implementation is accurate to 2^{-10}, which can easily be verified using exhaustive testing as the input space only contains 2^8 values.

$$|\ln(1+x) - \overline{\ln(1+x)}| < 2^{-10} \qquad (x = 0.x_0...x_7)$$

The first step is to generate the template problem. In this problem, the upper bits are represented by a constant, $\_\_y = 0.x_0...x_3$, and the lower bits, X, are modelled as a MetiTarski variable, to which we assign a specified range. The coefficients, which will be replaced in step 2 of the procedure, are just placeholders,

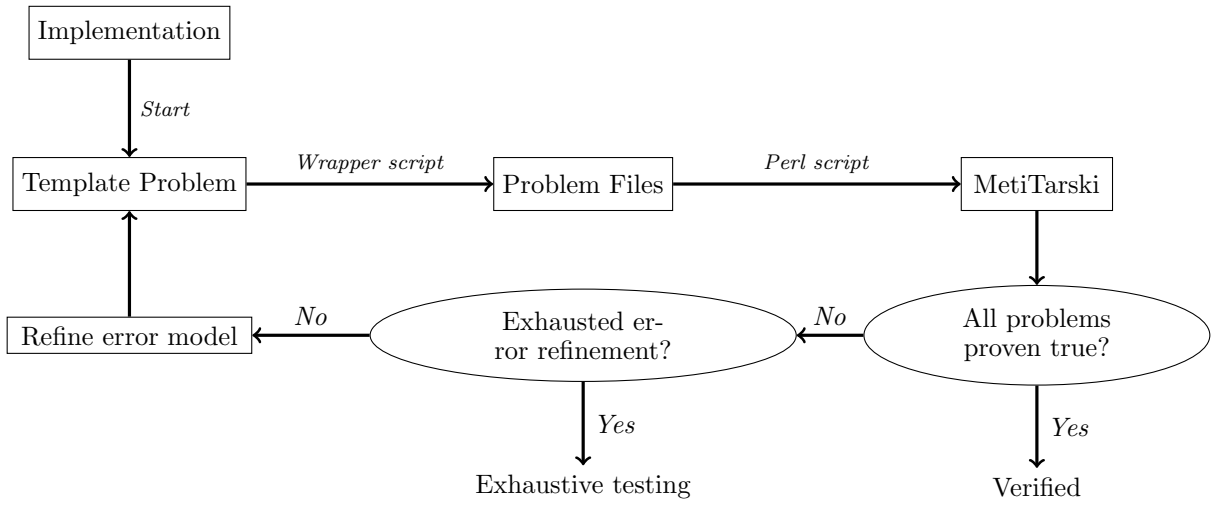


Fig. 2. Flow diagram of the verification procedure. Always start by generating an initial template problem from the given implementation.

$\_\_a$, $\_\_b$ and $\_\_c$. The $\Rightarrow$ should be read as implies. If this were a real hardware implementation the design could be improved by absorbing $\_\_y$, a constant, into the pre-calculated coefficients, resulting in a polynomial just in X.

$$\forall X \in [0, 2^{-4} - 2^{-8}] \Rightarrow |\ln(1 + \_\_y + X) - (\_\_c + \_\_b(\_\_y + X) + \_\_a(\_\_y + X)^2)| < 2^{-10}$$

This formula is the template problem which contains a single variable, X, and four placeholders, $\_\_y$, $\_\_a$, $\_\_b$ and $\_\_c$. A wrapper script now generates the 16 MetiTarski problems, replacing the placeholders $\_\_a$, $\_\_b$ and $\_\_c$ with the actual coefficients from the LUT and $\_\_y$ with the relevant constant input to the LUT. A Perl script, which is supplied with MetiTarski, automates the calls to our prover, providing a true or false result for each problem. For our toy implementation, MetiTarski is able to provide proofs for all of these problems and therefore steps 4 and 5 of the procedure are rendered redundant. Next we enhance the toy implementation and start to see where the refinement step is useful. The total runtime was 5.3 seconds and no more than 0.4 seconds is spent on any one proof. On such a small input space, exhaustive search is quicker by several orders of magnitude, taking less than a tenth of a second. However, as we shall see, our technique does not suffer from exponentially increasing runtimes as we increase the precision of the implementation.

With this basic understanding of the methodology, we shall make the toy implementation more realistic. Commercial hardware engineers have constraints on area and performance targets to meet, so they apply techniques to reduce the resource requirements. One of these is to truncate bits throughout the algorithm, reducing the number of adders required. This generally improves the performance but typically with some cost to the accuracy of the approximation. In our implementation, we choose appropriate terms to truncate in order to more closely replicate commercial algorithms. The new implementation returns an approximation of the form,

$$\overline{\ln(1 + x)} = c + 2^{-8}\lfloor b(x_0...x_7)\rfloor + a(0.x_0...x_5)^2. \tag{3}$$

In this case, the approximation is accurate to 2^{-7}.

$$|\ln(1 + x) - \overline{\ln(1 + x)}| < 2^{-7} \qquad (x = 0.x_0...x_7)$$

This can easily be checked using exhaustive testing, but notice that the implementation uses bit truncation on the first and second order terms. Since MetiTarski has no understanding of the integers, such non-analytic functions are difficult to model. In HOL Light, Harrison developed an entire theory of floating point arithmetic to more closely model the hardware he intended to verify [Har99]. For our purposes, it was sufficient to explore only simple approximations and bounds to such arithmetic. Inspired by this research, MetiTarski now includes support for a floor function. MetiTarski understands functions via axioms that give

upper and lower bounds, and in the case of the floor function we simply have $x-1 < \text{floor}(x) \leq x$. It should be noted that this bound is poor when the inputs under investigation are close to 1, for example MetiTarski will fail to prove $\text{floor}(0.5) \geq 0$. A simple extension could be the introduction of a bit truncation function, that takes an input x as well as the number of bits to truncate y. This would yield bounds:

$$x - (1 - 2^{-y}) \leq \text{trunc}(x, y) \leq x. \tag{4}$$

However, for the examples investigated in this paper, the basic floor function is sufficient to verify the function to the same precision that can be verified via exhaustive testing. Therefore, using this function, it is possible to produce a MetiTarski conjecture. We now allow X to be the integer value of the bottom 4 bits ($x_4x_5x_6x_7$) and $\_\_y$ the integer value of the top 4 bits ($x_0x_1x_2x_3$), which, as before, determines the coefficients $\_\_a, \_\_b, \_\_c$. We now use the integer values of X and $\_\_y$, because doing so allows us to model bit truncation using the floor function. Our new template problem is then,

$$0 \leq X \leq 15 \Rightarrow |\ln(1 + 2^{-4}\_\_y + 2^{-8}X) - \overline{\ln(1+x)}| < 2^{-7} \qquad (x = 0.x_0...x_7)$$

where,

$$\overline{\ln(1+x)} = \_\_c + 2^{-8}\,\text{floor}(\_\_b(X + 2^4\_\_y)) + 2^{-12}\_\_a(\text{floor}(2^{-2}X + 2^2\_\_y))^2.$$

Using the floor function provides a simple, but usually effective, model. However, this approach has an issue, which it shares with interval arithmetic, in that all correlation of errors is lost by this approach. To demonstrate this problem consider the following equation and MetiTarski floor function model of it, where z is a 4 bit integer.

$$(z >> 1) - (z >> 3) \rightarrow \text{floor}(\frac{1}{2}z) - \text{floor}(\frac{1}{8}z)$$

$$\text{floor}(\frac{1}{2}z) \in [\frac{1}{2}z - 1, \frac{1}{2}z], \quad \text{floor}(\frac{1}{8}z) \in [\frac{1}{8}z - 1, \frac{1}{8}z]$$

The floor function model indicates that this equation is bounded below by $\frac{1}{2}z - 1 - \frac{1}{8}z = \frac{3}{8}z - 1$. In actual fact the equation is bounded below by $\frac{3}{8}z - \frac{3}{8}$. This discrepancy is a result of disregarding any correlation between the two terms in the model and modelling it as a floor function rather than truncation. This issue also occurs in interval arithmetic, but fortunately we can deploy additional variables in our model which account for some of this correlation. By using two additional error variables our MetiTarski problem can model this behaviour more accurately.

$$\begin{aligned}(z >> 1) - (z >> 3) \rightarrow \frac{1}{2}z - \epsilon_0 - (\frac{1}{8}z - \epsilon_1 - \frac{1}{4}\epsilon_0) \qquad & z \in [0, 15], \quad \epsilon_0 \in [0, \frac{1}{2}], \quad \epsilon_1 \in [0, \frac{3}{4}] \\ = \frac{3}{8}z + \epsilon_1 - \frac{3}{4}\epsilon_0 \in [\frac{3}{8}z - \frac{3}{8}, \frac{3}{8}z + \frac{3}{4}] & \end{aligned}$$

Essentially, the error variable, ϵ_0 bounds bit 0 and ϵ_1 bounds a chunk of two bits, bits 1 and 2, of the variable z, allowing us to attain the bound we found analytically above, tighter than the bound we found using the floor function approach. Clearly, if we were to introduce an error variable for every bit or collection of bits truncated, then the number of variables in our MetiTarski problem would grow to be huge for any real world design. This is problematic as MetiTarski runtimes can be doubly exponential in the number of variables, an inherent limitation of the decision procedure on which it relies. In addition, this approach to error modelling requires significantly more user input and skill than the simple floor function method. Intellectual effort is needed to calculate the correlation between error terms introduced in the hardware algorithm. We can also model errors more carefully using a truncation model (eqn. 4) rather than the floor function approach to improve the tightness of the error bounds.

This example is intended to highlight the limitations of the floor function method, and to demonstrate that with additional human and computational effort it is possible to refine our error models. Returning to the procedure outline given above, the floor function method should be used to generate the initial template problem in step 1. If in step 3, MetiTarski fails to prove all the problems it is given, we can introduce error variables in our template problem to refine our error model. This is typically an iterative approach, as there is a tradeoff between the human effort required and the tightness of the error bounds. So there may be several rounds of error model refinement with MetiTarski runs until either, the problems are all proven or the error model refinements are exhausted. If, after exhausting all error model refinements, some

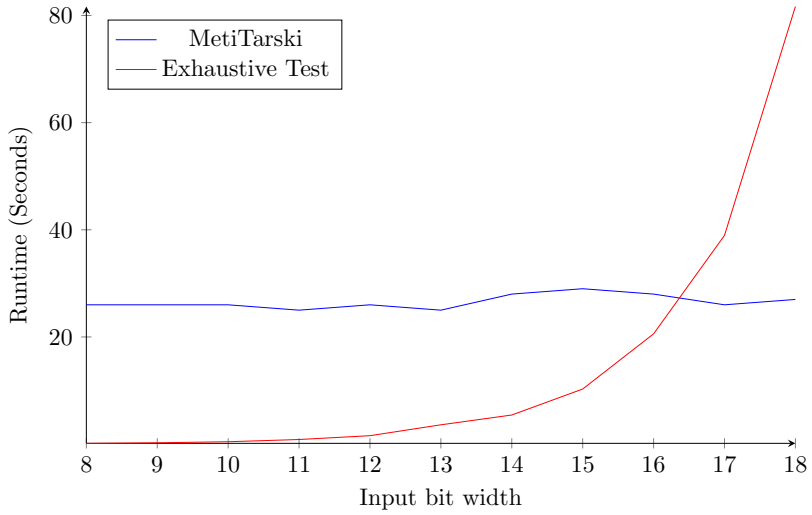


Fig. 3. A graph demonstrating the runtime comparison of the competing verification procedures on implementations of growing precision, results obtained running on a single core Intel I7-3517U

of the MetiTarksi problems remain unproven, then the last step is to use exhaustive testing on the remaining regions. All this effort is typically not wasted, as proving a subset of the MetiTarski problems will reduce, possibly significantly, the size of the input space left for verification using exhaustive testing. In §4, we discuss a complex industrial implementation where this final exhaustive step was necessary to complete the verification. This is the only case where we required exhaustive testing.

To illustrate this approach, we model our updated implementation, eqn. 3, using error variables rather than the floor function. This yields a new MetiTarski problem,

$$0 \leq X \leq 15 \ \wedge \ 0 \leq \epsilon_0 < 1 \ \wedge \ 0 \leq \epsilon_1 < 1 \Rightarrow |\ln(1 + 2^{-4}\_\_y + 2^{-8}X) - M\_\ln(1 + x, \epsilon_0, \epsilon_1)| < 2^{-7}$$

where,

$$M\_\ln(1 + x, \epsilon_0, \epsilon_1) = \_\_c + 2^{-8}(\_\_b(X + 2^4\_\_y) - \epsilon_0) + 2^{-12}\_\_a((2^{-2}X + 2^2\_\_y) - \epsilon_1)^2.$$

Surprisingly, for this particular problem, using additional error variables rather than the floor function actually had minimal impact on the overall runtime for the 16 problems. However, the floor function is a recent addition to MetiTarski and we may be able to improve its performance by fine-tuning its heuristic parameters. On industrial implementations, to limit the number of variables it was sometimes necessary to combine error variables and manually calculate an upper bound on these. This was often challenging: some of the error variables can be highly correlated. The floor function method makes the process of generating an initial template problem significantly simpler. For the industrial implementations verified in this paper, error variables were necessary, since tight error bounds were required.

To see why this technique is powerful, we extend the implementation above to larger input spaces. The approximation is essentially the same, using the same coefficients and lookup table, but our input now may be 10 bits rather than 8 bits, for example. Figure 3 compares the runtimes of our methodology and exhaustive testing as the input space grows. Notably, the MetiTarski method has roughly constant runtimes, as we expect: MetiTarski is an analytic tool, so increasing the space of discrete inputs only minimally alters the MetiTarski problems. Conversely, exhaustive testing runtimes suffer from exponential growth in the number of bits of the input. Of course, if the size of the lookup table increases, this will affect the MetiTarski runtimes as the number of problems will grow exponentially. The tables used in these algorithms are typically not prohibitively large, those referenced in §2 contained less than 256 entries, since large tables translate into additional silicon in hardware designs. We have already highlighted the inverse relationship between LUT size and polynomial degree in Figure 1, which suggests that higher precision implementations can use higher degree polynomials to limit LUT size growth. The toy example problem could be applied to any elementary function as the architecture framework, relying on piecewise quadratic interpolation, is function independent.

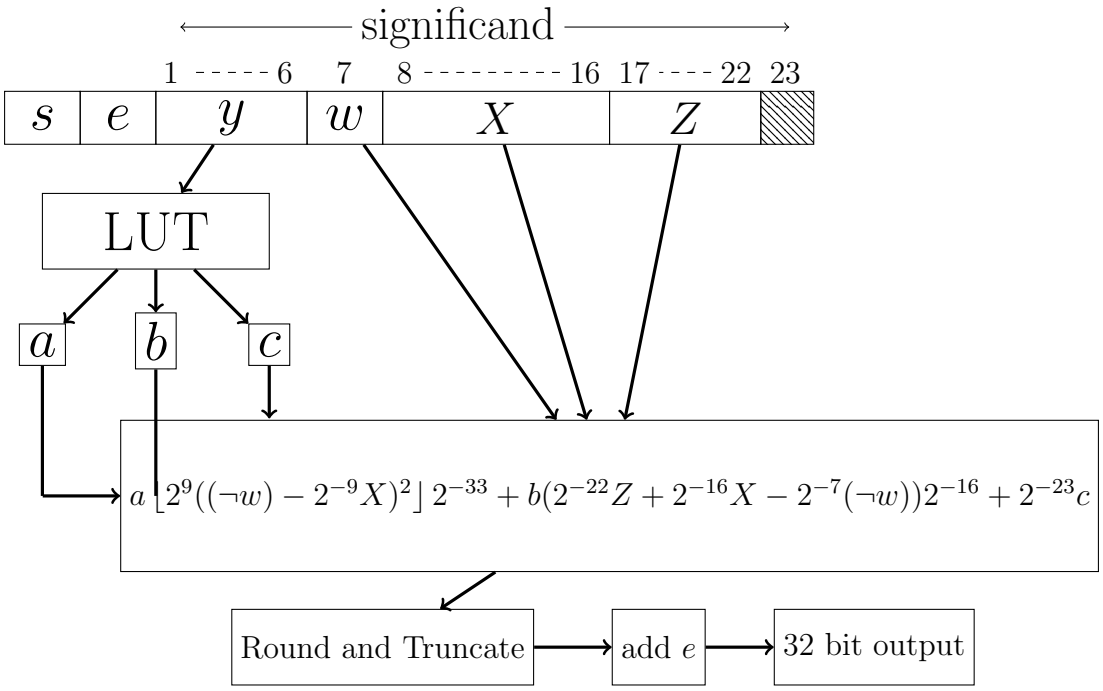


Fig. 4. A description of the implementation of the floating point $\log_2$ which MetiTarski was used to verify. The input is a 32 bit floating point number where s is the sign bit, e is the exponent, y is the top 6 bits of the significand and w ($\neg w =!w$), X and Z are divisions of the significand. The last bit is discarded.

4. Applications and Discussion

We present some results from applying this methodology to the verification of several larger commercial implementations. The runtime results can be found in Table 1 and demonstrate that the technique has real world relevance.

Figure 4 gives a description of the floating point implementation verified using this methodology. This implementation was a release candidate for Cadence Design Systems. We see that the 23 bit significand is split into 4 used sections, whilst the last bit is discarded. The top 6 bits, y, are passed to the LUT which outputs 3 polynomial coefficients. Using these coefficients, the algorithm computes a polynomial approximation to the logarithm. The split significand is used to reduce the number of operations along with a truncation of bits which are insignificant for the accuracy target. The accuracy target for the implementation was 4 ULPs [MBDD$^+$10], which is why, for the following conjectures the bounds on the distance from the true logarithm are $2^{-21}(= 4 \times 2^{-23})$, since we consider inputs in the region [1,2), as justified by equations 1 & 2. The exponent, e, in our input region is equal to the IEEE-754 bias b, yielding a zero exponent in the decoded value. This was the first implementation verified using the MetiTarski method, and as a result, when this was investigated the floor function was not supported. However, rather than calling the floor function explicitly the bounds on the calculations were calculated by hand and combined to yield a simple bound on the error in either direction. This is a labour intensive method, and was the inspiration behind the introduction of the floor function. This manual approach, resulted in two separate template problems.

$$(l = 0.693147180559945309417232121458176 \ \& \ 0 \leq X \leq 2^9 - 1 \ \& \ 0 \leq Z \leq 2^6 - 1) \Rightarrow$$

Template 1

$$2^{-33}a\Big(2^9(1-w) + 2X(w-1) + 2^{-9}X^2\Big) + 2^{-38}b\Big(Z + 2^6X + 2^{15}(w-1)\Big) + 2^{-23}c - \frac{1}{l}\ln(1 + y + 2^{-7}w + 2^{-16}X + 2^{-22}Z) \leq 2^{-21} + 2^{-33}a$$

Template 2

$$2^{-33}a\Big(2^9(1-w) + 2X(w-1) + 2^{-9}X^2\Big) + 2^{-38}b\Big(Z + 2^6X + 2^{15}(w-1)\Big) + 2^{-23}c - \frac{1}{l}\ln(1 + y + 2^{-7}w + 2^{-16}X + 2^{-22}Z) \geq -2^{-21} + 2^{-23}$$

Upper case letters are variables with defined ranges, which correspond to the different sections of the significand described in Figure 4. The lower case letters are constants in each problem, which are replaced by the python wrapper script, where in particular $w \in \{0, 1\}$, is a single bit of the significand as shown in the diagram. We note here that the splitting of the significand has forced the use of two variables, X and Z, to model the problem. On the right hand side of the inequality in Template 1, the $2^{-33}a$ is an error term introduced to model the truncation of the squared term in the algorithm. More precisely, we are explicitly using the naive floor function bound,

$$\left\lfloor 2^9((\neg w) - 2^{-9}X)^2 \right\rfloor > (2^9((\neg w) - 2^{-9}X)^2 - 1). \tag{5}$$

In Template 1 we are trying to prove an upper bound on the approximation. We want to make the square term as small as possible so use its lower bound, since a is negative. This should not be surprising since the coefficient of the square term in the Taylor expansion of $log_2(1 + x)$ is negative. We move the error term to the right hand side of the inequality for readability.

In these conjectures, l, is a high precision approximation to ln(2), which is necessary because MetiTarski only supports the natural logarithm, meaning we need to switch the base. Generated using MPFR [FHL+07], it was truncated at the 32nd decimal place. Therefore $0 < \ln(2) - l < 10^{-32}$. This error can be accounted for by the over-approximation of the truncation error term described by equation 5. This lower bound can actually be made tighter as we only truncate off at most 9 bits, so becomes

$$\left\lfloor 2^9((\neg w) - 2^{-9}X)^2 \right\rfloor \geq (2^9((\neg w) - 2^{-9}X)^2 - (1 - 2^{-9})).$$

This means that we have over-approximated the error by $2^{-42}a$. The error introduced by using l is

$$\frac{1}{l}\ln v - \frac{1}{\ln 2}\ln v = \frac{\ln 2 - l}{l \times \ln 2}\ln v < \frac{10^{-32}}{l \times \ln 2}\ln v.$$

For the cases we are interested in $|\ln v| \leq 2$ and $|a| \geq 10^{-1}$ in all the polynomials. This means that the error introduced by the l approximation is less than $2^{-42}a$ and is accounted for in Template 1.

Of course, Template 1 only proves half of the verification problem. We must also prove the other side, Template 2, which we can AND with the first, allowing us to keep the same coefficients and variable bounds. The only difference is that we check the lower bound and also calculate a new error bound. In this case, the 2^{-23} term on the right hand side of the inequality is to account for the final truncation and rounding of the output in order to fit it into floating point format. It relies on the same floor function bound used in equation 5, but applied to the whole polynomial rather than a single term. It has the opposite sign to the previous error term and is moved to the other side of the inequality once again. There is no need to account for the error in l in this template as $\ln(2) - l > 0$. If we can satisfy both these conjectures then we have obtained a proof for the given input range, in our case [1,2).

An error variable, $\epsilon \in [2^{-33}a, 2^{-23}]$, could have been used to reduce this to just one template problem, however testing showed that it was more efficient to split the conjecture in two to avoid the additional variable. The proofs of these problems and minor variants for inputs in the region [0.5,1) were obtained successfully with runtimes presented in Table 1. With a LUT containing 32 entries, the total number of

Design	Floating point $\log_2$	Fixed point 8.24 $\log_2$
Input Region Verified	$[0.5, 2)$	$[1, 2)$
MetiTarski Problems	128	512
MetiTarski Time (mins.)[a]	7	4
Exhaustive Test Time (mins.)[b]	42	53

Table 1. Verification runtime results for industrial implementations of logarithm base 2. The floating point implementation is a binary 32 bit IEEE-754 implementation.
a. Run on Intel Xeon E5 2698, speed: 2.3 GHz, CPUS: 2, cores: 2, cache-size: 40MB
b. Run on Intel Xeon X5680 machines, speed: 3.33 GHz, CPUS: 2, cores: 12, cache-size: 12MB

MetiTarski problems for this verification task was 128 ($= 32 \times 2 \times 2$), since each entry gave two problems and we needed slightly modified templates for the [0.5,1) region.

Previously, the implementation had been checked using a basic benchmarking tool. Through our investigations, we discovered that the expected accuracy of the implementation was breached for a small input region. Since exponent scaling for $\log_2$ simply involves adding the exponent to the approximation made on the 1.*significand* component, a bespoke and simple proof was sufficient to complete the verification. However, the proof identified a region in which the claimed accuracy may be breached and exhaustive testing on this region identified a large number of counterexamples. For the purpose of validating our methodology, we exhaustively tested all inputs: as predicted, everywhere but in this narrow region the implementation met its accuracy claim.

The second implementation in Table 1 is an experimental test case generated primarily to analyse how the MetiTarski method performed on higher precision algorithms. It takes as input a fixed point number with 8 bits before the decimal place and 24 bits after. Therefore, the total number of possible inputs is 2^{32}, but it is a higher precision algorithm and is more accurate as it makes use of a larger lookup table (256 entries). The algorithm used in this implementation is very similar to the one presented in Figure 4, taking a fixed point input, therefore we shall not discuss the details of the implementation. The template problem files were also extremely similar, if not simpler, but the main focus of this analysis was to test how the technique performed on higher precision algorithms, as the claimed accuracy of this implementation was 2^{-28}. We found that the technique was equally successful in verifying this implementation.

We see that the speedup achieved by MetiTarski is significant on the floating point logarithm. Notably it was even greater for the higher precision (8.24 fixed point) experimental implementation. Such an observation suggests that this methodology could be viable for verification of higher precision hardware, where traditional exhaustive techniques are infeasible. Evidence supporting this claim will be given in §5.

In addition to the results above, we applied our technique to a more complex algorithm, which had already been formally verified. Here we were only able to prove a subset of the problems. If we were attempting to verify a new implementation this outcome is still helpful as we reduce the space of inputs on which exhaustive testing is necessary. Across our experiments this was the only case where we found it necessary to resort to exhaustive testing. Through this experience we discovered a significant shortcoming of the approach. Generating the initial template problem is a non-trivial task. The verification engineer needs a deep understanding of the design implementation in order to model the errors correctly. Floor functions may be used initially to establish whether the algorithm is correct to a higher error rate, but to obtain a full proof, several refinements of the model may be required. Such refinements should take into account correlations in the error terms for the different sub-intervals. This aspect of the technique becomes time consuming when trying to verify more involved algorithms. For example, a sequence of **if** and **else if** statements, which can be found in some hardware algorithms, and much more commonly in software, are difficult to model using this technique. By comparison, exhaustive testing is far simpler to implement: it requires only knowledge of the output and the correct answer, which we may reasonably assume we have given the existence of the MPFR library [FHL+07]. This fact currently limits the circumstances under which our method is useful. For example, iterative algorithms for division and square root which use a variable number of iterations depending on the input are likely intractable.

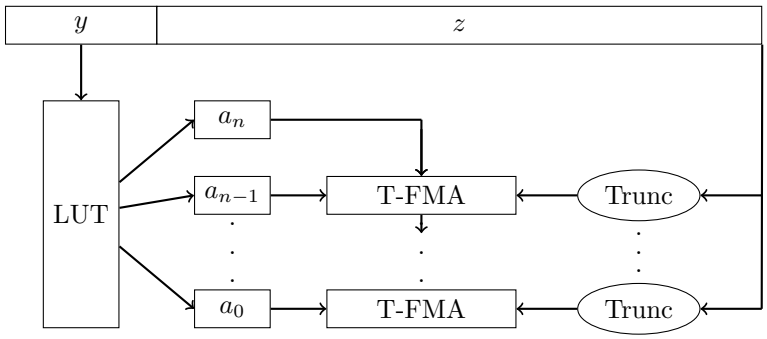


Fig. 5. Architecture of the *FixFunctionByPiecewisePolynomial* option generated by FloPoCo [dDJP10]. The T-FMA units are truncated fused-multiply adds, implementing $(\alpha \times \beta + \gamma)$. The Trunc operators just truncate the bits of z as not all of them are required for the FMA operation.

5. Automatically Generated Designs: FloPoCo

An additional source of implementations we can verify is the FloPoCo tool [dDP11]. FloPoCo can generate arithmetic cores for a range of applications, with its primary focus being on FPGAs. Its operators are fully parameterisable in precision and can be given error specifications to meet. The tool outputs synthesisable VHDL, which is generally human readable. It is still actively developed and used by the academic community, winning the *Community Award* of FPL 2017. FloPoCo has a wide range of methods and operators that it can generate, but we will focus on just one of these.

The *FixFunctionByPiecewisePoly* option generates a piecewise polynomial VHDL implementation for a given function $f(x)$, where $x \in [0, 1]$ and has a user specified number of bits. The user specifies the degree n of the polynomial that will be used in the approximation. By default, it generates an approximation to $f(x)$, which is claimed to be correct to within 1 ULP.

The authors [dDJP10] have described their method for generating approximations. They generate polynomial coefficients for each interval, where each interval spans the same input width with the most significant bits of the input x determining which polynomial is used. This is implemented as a LUT with one entry per polynomial/interval and width determined by the combined width of the coefficients. One of the benefits of using FloPoCo is that the user does not need to specify the number of intervals to split the domain into. The tool finds a number of intervals for which it can find a set of polynomials of degree n that meet the error specification, allowing some of the error budget for rounding errors in implementing the polynomial evaluation. The authors do not claim that this is the smallest number of polynomials that would achieve the error budget.

Generating the polynomial coefficients is done using a modified Remez algorithm [BC07], that essentially computes minmax polynomials over the space of polynomials with coefficients with finite precision. This removes the need to round real coefficients, avoiding additional error. The Sollya tool [CJL10] handles the polynomial generation and provides an error bound.

Polynomial evaluation is then implemented using the Horner evaluation scheme,

$$\begin{aligned} p(x) &= a_n x^x + a_{n-1} x^{n-1} + \cdots + a_0 \\ &= (\cdots (a_n x + a_{n-1}) x + \cdots + a_1) x + a_0. \end{aligned}$$

Internal truncation is used in this implementation because $x \in [0, 1]$ implies that there is no need to compute $a_n x^n$ to full precision: most of its bits will not be significant. The architecture of the implementation is described by Figure 5, which motivates the error modelling in our MetiTarski problems.

We will verify the authors' 23 bit and 52 bit examples [dDJP10], which use degree 2 and degree 4 polynomials, respectively, to approximate $\sqrt{1+x}$. This is equivalent to computing $\sqrt{1.sig}$ for a binary64 input, where the exponent scaling may be handled by a different algorithm. Table 2 shows some details of the implementations and the verification effort required.

To illustrate the method applied to this example, consider just the 52 bit case. The LUT contained 256

Bits	Polynomial Degree	LUT Entries	Problems	Variables	Total Time (min)	Average Time (sec)
23	2	64	128	2	2.4	1.13
52	4	256	1024	3	180	10.8

Table 2. Verification results for FloPoCo generated square root implementations.

entries corresponding to 256 different polynomials. We extract the coefficients table from the VHDL and use a wrapper script to insert these values as well as the most significant bits of the input into the template problem, as described in the general methodology. The implementation uses 4 truncated multiply add blocks. The truncation of the remaining input bits z for each of these, as shown in Figure 5, forces us to split z into 4 MetiTarski variables:

$$z = \boxed{z_1}\boxed{z_2}\boxed{z_3}\boxed{z_4}$$

$z_1 \in [-2^{-9}, 2^{-9} - 2^{-27}]$, $z_2 \in [0, 2^{-27} - 2^{-36}]$, $z_3 \in [0, 2^{-36} - 2^{-46}]$, $z_4 \in [0, 2^{-46} - 2^{-52}]$.

To centre the interval in the VHDL implementation the z_1 variable is treated as signed. In addition to the truncations on the input bits to the FMAs they are also truncated on the output, which we model as errors E_i for each FMA. With an error specification that says we should be within 1 ULP of the exact square root, this yields an initial MetiTarski problem of the form

$$\begin{aligned}|\sqrt{1+y+z_1+z_2+z_3+z_4} - ((&A_4z_1 + A_3 - E_4)(z_1+z_2)\\ &+A_2 - E_3)(z_1+z_2+z_3)\\ &+A_1 - E_2)(z_1+z_2+z_3+z_4)\\ &+A_0 - E_1) - E_0| < 2^{-52}\end{aligned}$$

Maximising the total error, and splitting based on z_1 positive or negative, splits this initial template into 4 problems. Unfortunately, MetiTarski was unable to solve this 4 variable problem, which is close to the limit of how many variables MetiTarski can handle.

We solved this problem by only using 3 variables, essentially introducing $z_3' = z_3 + z_4$. Then wherever we want to use just z_3, we can minimise/maximise the value of z_4 and use the bounds $z_3' - (2^{-46} - 2^{-52}) \leq z_3 \leq z_3'$. Splitting the problems based on z_1 positive and negative and minimising/maximising the error in the polynomial evaluation yields 4 problems per interval/polynomial. Since our output has 52 bits after the decimal place, 1 ULP corresponds to a total error bound of 2^{-52}. The 23 bit example uses a similar modelling approach but only needs two problems per interval/polynomial.

The results are summarised in Table 2, where we see that each 52 bit problem takes 10× longer to solve than each 23 bit problem on average. This is due to the extra variable and the tighter error bound as well as the increased complexity due to the higher polynomial degree. A 3 hour verification runtime is certainly reasonable for a binary64 implementation, which clearly cannot be exhaustively tested.

6. A Comparison with Gappa

Having given several examples of how our approach can be used, we shall now look at a comparison with an existing tool. For verification of small floating point functions, the Gappa tool [DDLM06, BFM09] has been tested and is still being developed. Gappa uses interval arithmetic in order to prove conjectures whilst MetiTarski relies on cylindrical algebraic decomposition to generate proofs. Both methods are valid; interval arithmetic is weaker but considerably more efficient. The approach using the Gappa tool, is the most comparable method to that described in this paper, therefore we reproduced the verification of the polynomial approximation to the sine function implemented as part of the CRlibm project [LDDD+09, dLM11]. The function verified here is a binary64 software implementation using *round to nearest*, which demonstrates not only the versatility of MetiTarski, but also shows that the axioms bounding the sine function are sufficiently accurate to verify high precision implementations.

The implementation verified here does not use lookup tables, so is a simpler case than we have been considering in the previous sections. In essence we can think of this as just using a single table entry to

yield the polynomial coefficients for all inputs, rather than sub-dividing the input domain. As a result the verification method deployed here uses a simpler approach than that described by Figure 2, eliminating the need for template problems, wrapper scripts or hundreds of MetiTarski calls.

The following C code quoted from [dLM11], is compiled respecting the C99 and IEEE-754 standards, with all variables being binary64.

```
s3 = -1.6666666666666665741e-01; // approximately -1/6
s5 =  8.3333333333333332177e-03; // approximately  1/120
s7 = -1.9841269841269841253e-04; // approximately -1/5040

yh2 = yh * yh;
ts  = yh2 * (s3 + yh2 * (s5 +yh2 * s7));
Fast2Sum(sh, sl, yh, yl + yh * ts);
```

In this code, the input is represented as a sum of two binary64 arguments, y = yh+yl, where yh represents the most significant part and yl the least significant, and the result, s, is similarly represented as a sum sh+sl. The Fast2Sum algorithm provides an exact representation of the sum of two floating point numbers as a pair of floating point numbers, namely no rounding errors are incurred. This approximation to sine is only valid for a limited input region, namely values of magnitude less than 6.3×10^{-3}. The code is an approximation to the Taylor expansion of sine, where the terms involving the least significant bits are discarded since they are sufficiently small.

$$\sin(y) = y - \frac{y^3}{6} + \frac{y^5}{120} - \frac{y^7}{5040} + ... \tag{6}$$

The Gappa approach makes use of the "IEEEdouble" shortcut for the IEEE-compliant binary64 *round to nearest* mode, in order to encode the rounding errors in this implementation. Since the IEEE-754 standard imposes exact arithmetic in binary64 *round to nearest* mode, the maximum relative error incurred in each arithmetic operation is 2^{-53}, since binary64 uses a 52 bit significand. This keeps the error modelling for our MetiTarski problems far simpler than in some hardware algorithms, which can operate at the bit level. Essentially, for each of the operations in the C code, a maximum relative error of 2^{-53} can be incurred. This property is commonly described as the $(1 + \epsilon)$ lemma. Harrison describes the bounds on ϵ in [Har06].

The full Gappa script is given in [dLM11], and for the purpose of comparison we use the same outline of the problem to prove, just using more explicit error modelling. They also introduce the bound on the input variable $|Y| \geq 2^{-200}$, which is necessary to make sure that the relative errors due to underflow remain bounded. In the Gappa script, 4 variables are used, since the sine is represented by a variable $S \in [0, 1]$. MetiTarski supports axiomatic bounds on functions such as sine, so this variable is dropped in order to reduce the problem to 3 variables. As Gappa has no notion of sine, first the approximation error between the exact polynomial approximation and sine had to be calculated. This approximation error takes no account of any floating point rouding errors. For this purpose the Sollya tool [CJL10], was used to prove that,

$$|\mathrm{Y}| \leq 6.3 \times 10^{-3} \Rightarrow \left| \frac{\mathrm{PolySinY} - \mathrm{SinY}}{\mathrm{SinY}} \right| \leq 3.7 \times 10^{-24}. \tag{7}$$

PolySinY represents the exact polynomial specified in the C code above with no floating point rounding errors. In MetiTarski, no additional tools were required, as the inbuilt axiomatic bounds on the sine function were sufficient. The variable Y, is the sum of our two binary64 variables H (msb) and L (lsb), with a bound on the relative error in the argument reduction stage, also present in the Gappa paper.

$$\left| \frac{H + L - Y}{Y} \right| < 2.53 \times 10^{-23} \quad \wedge \quad |L| < 2^{-53}|H|.$$

The goal is to prove that the maximum relative error in our approximation of $\sin(Y)$, is less than 2^{-67}. Since we explicitly calculate the error modelling, and have used placeholder replacement as a strategy throughout the previous sections, the template problems include error placeholders e_i for $i = 0, ..., 7$, where each $e_i \in [1 - 2^{-53}, 1 + 2^{-53}]$, representing the relative error in each arithmetic operation. With these errors

inserted the polynomial approximation to sine after expanding out all the brackets reduces to

$$f(H, L, e_0, ..., e_7) = H + e_7(L + g(H, e_0, ..., e_6)).$$
$$g(H, e_0, ..., e_6) = e_0e_4e_5e_6 \times aH^3 + e_0^2e_2e_3e_4e_5e_6 \times bH^5 + e_0^3e_1e_2e_3e_4e_5e_6 \times cH^7$$

Ideally we would introduce a new MetiTarski variable for each e_i, however MetiTarski is unlikely to terminate when using more than 4 or 5 variables, for the reasons described in [AP08], so we need to make choices to bound these errors. By considering only $H > 0$, and choosing appropriate values for $e_i = 1 \pm 2^{-53}$, we obtain

$$g_{\min}(H) \leq g(H, e_0, ..., e_6) \leq g_{\max}(H)\ \forall e_i \in [1 - 2^{-53}, 1 + 2^{-53}]. \tag{8}$$

We consider only $H > 0$, which implies that $\sin(Y) > 0$. We combine this with the constraints described above.

$$\left|\frac{f(H, L, e_0, ..., e_7) - \sin(Y)}{\sin(Y)}\right| < 2^{-67} \quad \forall e_i \in [1 - 2^{-53}, 1 + 2^{-53}] \quad \Leftrightarrow \tag{9}$$
$$f(H, L, e_0, ..., e_7) < (1 + 2^{-67}) \sin(Y) \ \wedge\ f(H, L, e_0, ..., e_7) > (1 - 2^{-67}) \sin(Y) \tag{10}$$
$$\forall e_i \in [1 - 2^{-53}, 1 + 2^{-53}]. \tag{11}$$

Using our bounding functions (eqn. 8), we can reduce the problem to proving the four inequalities

$$H + e_7(L + g_{\max}(H)) < (1 + 2^{-67}) \sin(Y) \tag{12}$$
$$H + e_7(L + g_{\min}(H)) > (1 - 2^{-67}) \sin(Y) \quad \text{for } e_7 = 1 \pm 2^{-53}. \tag{13}$$

Due to the error bounding approach and the anti-symmetry of sine and the polynomial in H, proving these four inequalities is sufficient to verify the original problem (eqn. 9).

MetiTarski required some minor assistance with a basic rewrite for one of the inequalities, just replacing the variable L with its relevant bound. The full proof scripts are given in Appendix A, where the final inequality takes a slightly different form due to the required rewrite. The 4 problems were provable using MetiTarski, with a combined runtime of 460 seconds on an Intel I7-3517U CPU.

We will briefly digress, as these experiments also highlighted some interesting MetiTarski behaviour. MetiTarski invokes an RCF solver during its operation, and it can be configured to use any one of Z3 [DMB08], QEPCAD B [Bro03] or Mathematica (invoked in batch mode) [Inc]. Given that Mathematica and Z3 are widely used in academia and industry we choose to trust their results. Only Mathematica terminates in a reasonable amount of time in this problem domain. In fact, using Z3 or QEPCAD B, MetiTarski hangs when trying to prove the significantly easier problem,

$$H + (1 - 2^{-53}) \times (L + g(H, 0, ..., 0)) < (1 + 2^{-67}) \sin(Y). \tag{14}$$

This introduces no errors in our function g. This discovery suggests that there could be further development in this field.

To conclude this section, we should compare the Gappa method to ours. Gappa's understanding of floating point rounding via the shortcut described above makes it simpler to generate problem scripts when analysing *pure* floating point algorithms. By *pure* we mean that the algorithm only uses standard floating point arithmetic with no bit level manipulations. These are common in software, such as the CRlibm sine function, but in hardware, designers rarely stick to *pure* floating point arithmetic, which would likely pose a different challenge for generating Gappa scripts. The Gappa approach had to involve other tools in its solution, whilst MetiTarski could be used in a standalone manner. With Gappa taking less than a second to generate a proof it clearly demonstrates stronger performance. However, Gappa required 6 additional "hints", with the authors describing the hint writing process as the most time consuming part. By comparison, MetiTarski required just one re-write of a problem in order to prove it, and no re-writes (or "hints") were required in our other examples, so MetiTarski is competing on an uneven playing field. In both approaches it is evident that writing the respective scripts is the time-consuming component rather than generating the proofs.

Given the availability of tools with inbuilt floating point rounding, such as Gappa, most software implementations will be more amenable to these tools rather than MetiTarski. However, this comparison has shown that the process of obtaining a proof involving an implementation of sine, using MetiTarski, is similar to the approach described for the logarithm earlier.

7. Related work

An important problem for verification will be the interplay of floating point and bit level operations, common in real code. As observed by de Dinechin, Lauter and Melquiond, expert developers will arrange code in order to take advantage of situations where floating point arithmetic is exact, such as multiplication by a power of two, to name just one [DDLM06]. Another trick, computing 2^n in double precision, can be done by shifting $n + 1023$ left by 52 bits [Min13]. Such techniques make verification more challenging, particularly in the context of high precision transcendental functions [LSA16]. Code obfuscation can easily occur in the optimisation stages, when the underlying mathematical approximations become obscured as expressions are manipulated and re-ordered in order to maximise floating point accuracy.

We have already described the Gappa approach, but many other tools have been applied to floating point algorithm verification. John Harrison conducted some of the earliest and most rigorous work via interactive theorem proving, using the HOL Light proof assistant [Har97]. Above, we highlighted how Harrison had formalised the theory of IEEE floating point arithmetic [Har99]. He defined the set of representable floating point numbers as a parameterised triple (E, p, N), which is the set of real numbers representable in the form $(-1)^s 2^{e-N} k$ with $e < E$, $k < 2^p$ and $s < 2$. Formalizations of the four standard rounding modes (nearest, down, up, zero), the widely used $(1 + \epsilon)$ lemma and other lemmas about exactness are constructed in HOL Light. As a result of this underlying theory, most floating point problems are side-stepped and he is able to reason about real numbers and straightforward algebraic calculations [Har99]. Such an approach is labour intensive and much of the labour is specific to the algorithm at hand. Note however, that Harrison's proofs are more detailed, including results about rounding modes.

The Coq system has also been the subject of several attempts to develop libraries [DRT01, Mel12, BM11] and was combined with Gappa [BFM09, DM10] in order to verify a wider range of floating point algorithms. As well as these, Boldo combined Coq with Caduceus for the verification of floating point C programs [BF07].

CoqInterval [MDM16] targets a very similar goal to the one presented in this paper. It is primarily concerned with verifying floating point mathematical libraries and as a result many of the problems look similar to ours, although there is less focus on piece-wise polynomial approximations. As a result CoqInterval is likely capable of proving many of the problems presented here. The key distinction is that CoqInterval is based on interval arithmetic, which is generally weaker than cylindrical algebraic decomposition (MetiTarski's underlying technology). The CoqInterval authors actually compared CoqInterval to MetiTarski, noting that MetiTarski was generally faster and could tackle more complex problems [MDM16]. However CoqInterval uses automated Taylor expansion, whilst MetiTarski relies on its built in axioms so is more restricted. Also CoqIntervals proofs are formally verified by Coq which may account for some of the performance difference.

FPTaylor is another recent tool, that bounds floating point round off errors again using Taylor expansion [SBB+18]. FPTaylor's benchmarks show that it is capable of tackling problems with up to 6 variables, with support for elementary functions. In addition to these studies several theorem provers have been applied to floating point verification problems, but in the cases referenced here additional libraries were developed to handle floating point numbers. The ACL2 prover was applied to floating point division, multiplication and square root algorithms [Rus98, MLK96]. Lastly the PVS prover was used to verify the floating point unit of a complete microprocessor [JB05]. In general, the main difference between this paper and those referenced above is that we target higher level algorithms for verification and don't attempt to encode floating point operations in MetiTarski explicitly.

8. Conclusion

We have successfully verified a variety of fixed and floating point logarithm, sine and square root implementations using the MetiTarski theorem prover. We have also identified a bug in an algorithm that was close to production. Most interesting are the binary64 results from §5, which suggest that this technique will be viable when higher precision algorithms are required. In a comparison with a more thoroughly explored technique, using Gappa, our approach held its own in successfully verifying a binary64 software implementation of the sine function. This highlights the potentially broad scope of this approach, although more complex software algorithms may not be amenable.

For a set of algorithms following a basic outline, it is possible to automate the whole verification procedure after the initial template and wrapper scripts are written. In this type of setting, the technique is quite powerful and to some degree function independent. However, for algorithms not following a standard format,

the verification process requires significantly more manual effort by a skilled verification engineer than alternative methods. For some cases, the approach may be unsuccessful if the algorithm uses techniques we can't model using MetiTarski. In all these examples the bulk of the time was spent manually error modelling with the MetiTarski proof times being relatively short, at most a few hours.

This first investigation into this technique has demonstrated both strengths and weaknesses, but further research may yield new methods for modelling more complex algorithms. Further development of MetiTarski could enhance its power or simplify the process, as the introduction of the floor function has already done. Continuing MetiTarski development to add new function bounds, such as for the base 2 logarithm would make future work simpler. A possible extension is the use of a binary chop applied to the input region of each problem, which may allow the engineer to further reduce the input space on which exhaustive testing is necessary. In this work there has generally been no verification of the argument reduction stage described in §2, so this may be an interesting topic to explore in the future. For implementations that follow a standard model and particularly bespoke higher precision implementations, the methodology shows promise.

Acknowledgements

We would like to thank Mathew Eaton and Ran Zmigrod for helpful contributions during the initial investigations. Much of the preliminary work for this paper was done at Cadence Design Systems, who were supportive throughout.

References

[And98] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 191–200, 1998.

[AP08] Behzad Akbarpour and Lawrence C Paulson. MetiTarski: An automatic prover for the elementary functions. In *International Conference on Intelligent Computer Mathematics*, pages 217–231. Springer, 2008.

[AP09] Behzad Akbarpour and Lawrence C. Paulson. Applications of MetiTarski in the verification of control and hybrid systems. In Rupak Majumdar and Paulo Tabuada, editors, *Hybrid Systems: Computation and Control*, LNCS 5469, pages 1–15. Springer, 2009.

[AP10] Behzad Akbarpour and Lawrence Charles Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[BC07] Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial L-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 169–176. IEEE, 2007.

[BF07] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE, 2007.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*, pages 59–74. Springer, 2009.

[BKM94] J-C Bajard, Sylvanus Kla, and J-M Muller. BKM: A new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, 43(8):955–963, 1994.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 243–252. IEEE, 2011.

[Bro03] Christopher W Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97–108, 2003.

[CGI+17] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In *International Conference on Automated Deduction*, pages 95–113. Springer, 2017.

[CHCK12] D. Chen, L. Han, Y. Choi, and S. Ko. Improved decimal floating-point logarithmic converter based on selection by rounding. *IEEE Transactions on Computers*, 61(5):607–621, 2012.

[CJL10] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[Cod80] William James Cody. *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. Prentice-Hall, Inc., USA, 1980.

[CZC+09] D. Chen, Y. Zhang, Y. Choi, M. H. Lee, and S. Ko. A 32-bit decimal floating-point logarithmic converter. In *19th IEEE Symposium on Computer Arithmetic*, pages 195–203, 2009.

[DAT+09] William Denman, Behzad Akbarpour, Sofiene Tahar, Mohamed H Zaki, and Lawrence C Paulson. Formal verification of analog designs using MetiTarski. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 93–100. IEEE, 2009.

[dDJP10] Florent de Dinechin, Mioara Joldes, and Bogdan Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[DDLM06] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using Gappa. In *ACM symposium on Applied computing*, pages 1318–1322. ACM, 2006.

[dDP11] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[DLM08] Marc Daumas, David Lester, and César Munoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, 2008.

[dLM11] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):2, 2010.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DRT01] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In *International Conference on Theorem Proving in Higher Order Logics*, pages 169–184. Springer, 2001.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13, 2007.

[Fow93] Raymond E. Fowkes. Hardware efficient algorithms for trigonometric functions. *IEEE Transactions on Computers*, 42(2):235–239, 1993.

[Gal91] Shmuel Gal. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software (TOMS)*, 17(1):26–45, 1991.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M Clarke. dreal: An SMT solver for nonlinear theories over the reals. In *International Conference on Automated Deduction*, pages 208–214. Springer, 2013.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[Har97] John Harrison. Floating point verification in HOL Light: the exponential function. In *International Conference on Algebraic Methodology and Software Technology*, pages 246–260. Springer, 1997.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 113–130. Springer, 1999.

[Har01] David Harris. A powering unit for an OpenGL lighting engine. In *Conference Record of Thirty-Fifth Asilomar Conference on Signals, Systems and Computers (Cat. No. 01CH37256)*, volume 2, pages 1641–1645. IEEE, 2001.

[Har06] John Harrison. Floating-point verification using theorem proving. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 211–242. Springer, 2006.

[Har09] J. Harrison. Decimal transcendentals via binary. In *19th IEEE Symposium on Computer Arithmetic*, pages 187–194, 2009.

[HKS+99] John Harrison, Ted Kubaska, Shane Story, et al. The computation of transcendental functions on the IA-64 architecture. In *Intel Technology Journal*. Citeseer, 1999.

[Inc] Wolfram Research, Inc. Mathematica, Version 12.1. Champaign, IL, 2020.

[JB05] Christian Jacobi and Christoph Berg. Formal verification of the VAMP floating point unit. *Formal Methods in System Design*, 26(3):227–266, 2005.

[LDDD+09] Catherine Daramy Loirat, David Defour, Florent De Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, and Jean-Michel Muller. CR-LIBM a library of correctly rounded elementary functions in double-precision, 2009.

[Lew95] David M Lewis. 114 MFLOPS logarithmic number system arithmetic unit for DSP applications. *IEEE Journal of Solid-State Circuits*, 30(12):1547–1553, 1995.

[LH07] Joe Leslie-Hurd. Metis theorem prover, 2007.

[LSA16] Wonyeol Lee, Rahul Sharma, and Alex Aiken. Verifying bit-manipulations of floating-point. *ACM SIGPLAN Notices*, 51(6):70–84, 2016.

[LW07] Michel Ludwig and Uwe Waldmann. An extension of the knuth-bendix ordering with LPO-like properties. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–362. Springer, 2007.

[MBDD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of Floating-Point Arithmetic*. Springer, 2010.

[MDM16] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016.

[Mel12] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[Min13] Antoine Miné. Abstract domains for bit-level machine integer and floating-point operations. In Jacques Fleuriot, Peter Höfner, Annabelle McIver, and Alan Smaill, editors, *ATx'12/WInG'12: Workshop on Invariant Generation*, volume 17 of *EPiC Series in Computing*, pages 55–70, 2013.

[MLK96] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5k86 floating point division algorithm. *URL: http://devil. ece. utexas. edu*, 80, 1996.

[PEB02] J. Pineiro, M. D. Ercegovac, and J. D. Bruguera. High-radix logarithm with selection by rounding. In *Proceedings IEEE International Conference on Application- Specific Systems, Architectures, and Processors*, pages 101–110, 2002.

[PEB04] J-A Pineiro, Milos D Ercegovac, and Javier D Bruguera. Algorithm and architecture for logarithm, exponential, and powering computation. *IEEE Transactions on Computers*, 53(9):1085–1096, 2004.

[POMB05] J-A Pineiro, Stuart F Oberman, J-M Muller, and Javier D Bruguera. High-speed function approximation using a minimax quadratic interpolator. *IEEE Transactions on Computers*, 54(3):304–318, 2005.

[Pra95] Vaughan Pratt. Anatomy of the Pentium bug. In *Colloquium on Trees in Algebra and Programming*, pages 97–107. Springer, 1995.

[Rus98] David M Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7™ processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[SBB+18] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):1–39, 2018.

[SDCP11] Antonio GM Strollo, Davide De Caro, and Nicola Petra. Elementary functions hardware implementation using constrained piecewise-polynomial approximations. *IEEE Transactions on Computers*, 60(3):418–432, 2011.

[SS93] M. Schulte and E. Swartzlander. Exact rounding of certain elementary functions. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 138–145, 1993.

[ST99] Shane Story and Ping Tak Peter Tang. New algorithms for improved transcendental functions on IA-64. In *Proceedings 14th IEEE Symposium on Computer Arithmetic*, pages 4–11. IEEE, 1999.

[Tan91] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. In *10th IEEE Symposium on Computer Arithmetic*, pages 232–236. IEEE, 1991.

[Vei60] L Veidinger. On the numerical determination of the best approximations in the chebyshev sense. *Numerische Mathematik*, 2(1):99–105, 1960.

[Vol59] Jack Volder. The CORDIC computing technique. In *Papers Presented at the March 3-5, 1959, Western Joint Computer Conference*, pages 257–261, 1959.

[Wal71] J. S. Walther. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, AFIPS '71 (Spring), page 379–385, New York, NY, USA, 1971. Association for Computing Machinery.

A. Sine Approximation Problem

We show the MetiTarski problems from §6, which uses 3 variables and the sin-extended axioms. There are 4 conjectures, which are subject to the same constraints. Variables are represented by capital letters and are declared within the square brackets. The constants are the result of compiling the C code from §6, adhering to the C99 standard. We shall present one script in full, then the other 3 just need to have the inequality to prove replaced.

- Y - the argument of sine
- H - the msb of Y (approx)
- L - the lsb of Y (approx)

```
fof(crlibm_sin, conjecture, ! [Y,H,L] :
((
a = -0.16666666666666665741480812812369549646973609924316406 25          &
b =  0.00833333333333333321768510160154619370587170124053955078125       &
c = -0.00019841269841269841252631711547849135968135669827461242675781 25 &
Y                                : (=2^(-200), 6.3e-3=)                   &
H                                > 0                                      &
L                                : (=-2^(-53)*H, 2^(-53)*H=)              &
H+L-Y                            : (-Y*2.53e-23, Y*2.53e-23)
)
=>
(
H + (L + ((1-2^(-53))^4 * a * H^3)
+ ((1-2^(-53))^5 * (1+2^(-53))^2 * b * H^5)
+ ((1-2^(-53))^7 * (1+2^(-53))^2 * c * H^7))*(1+2^(-53))
< (1+2^(-67))*sin(Y)
))).

include('Axioms/general.ax').
include('Axioms/sin-extended.ax').
```

Additional inequalities to prove:

```
H + (L + ((1-2^(-53))^4 * a * H^3)
+ ((1-2^(-53))^5 * (1+2^(-53))^2 * b * H^5)
+ ((1-2^(-53))^7 * (1+2^(-53))^2 * c * H^7))*(1-2^(-53))
< (1+2^(-67))*sin(Y)

H + (L + ((1+2^(-53))^4 * a * H^3)
+ ((1+2^(-53))^5 * (1-2^(-53))^2 * b * H^5)
+ ((1+2^(-53))^7 * (1-2^(-53))^2 * c * H^7))*(1+2^(-53))
> (1-2^(-67))*sin(Y)

H + L + (((1+2^(-53))^4 * a * H^3)
+ ((1+2^(-53))^5 * (1-2^(-53))^2 * b * H^5)
+ ((1+2^(-53))^7 * (1-2^(-53))^2 * c * H^7))*(1-2^(-53)) - 2^(-106)*H
> (1-2^(-67))*sin(Y)
```