HAL
archives-ouvertes.fr

Certifying the floating-point implementation of an elementary function using Gappa

Florent de Dinechin, Christoph Lauter, Guillaume Melquiond

To cite this version:

Florent de Dinechin, Christoph Lauter, Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers, Institute of Electrical and Electronics Engineers, 2011, 60 (2), pp.242-253. 10.1109/TC.2010.128 . ensl-00200830v2

HAL Id: ensl-00200830

https://hal.inria.fr/ensl-00200830v2

Submitted on 8 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Certifying the floating-point implementation of an elementary function using Gappa

Florent de Dinechin, *Member, IEEE,* Christoph Lauter, and Guillaume Melquiond

Abstract—High confidence in floating-point programs requires proving numerical properties of final and intermediate values. One may need to guarantee that a value stays within some range, or that the error relative to some ideal value is well bounded. This certification may require a time-consuming proof for each line of code, and it is usually broken by the smallest change to the code, e.g. for maintenance or optimization purpose. Certifying floating-point programs by hand is therefore very tedious and error-prone. The Gappa proof assistant is designed to make this task both easier and more secure, thanks to the following novel features. It automates the evaluation and propagation of rounding errors using interval arithmetic. Its input format is very close to the actual code to validate. It can be used incrementally to prove complex mathematical properties pertaining to the code. It generates a formal proof of the results, which can be checked independently by a lower-level proof assistant like Coq. Yet it does not require any specific knowledge about automatic theorem proving, and thus is accessible to a wide community. This article demonstrates the practical use of this tool for a widely used class of floating-point programs: implementations of elementary functions in a mathematical library.

Index Terms—Correctness proofs, error analysis, elementary function approximation.

1 Introduction

Floating-point (FP) arithmetic [1] was designed to help developing software handling real numbers. However, FP numbers are only an approximation to the real numbers. A novice programmer may incorrectly assume that FP numbers possess all the basic properties of the real numbers, for instance associativity of the addition, and waste time fighting the subtle bugs they induce. Having been bitten once, the programmer may forever stay wary of FP computing as something that cannot be trusted. As many safety-critical systems rely on floating-point arithmetic, the question of the confidence that one can have in such systems is of paramount importance, all the more as floating-point hardware, long available in mainstream processors, is now also increasingly implemented into embedded systems.

This question was partly addressed in 1985 by the IEEE-754 standard for floating-point arithmetic and its revision in 2008 [2]. This standard defines common floating-point formats (single and double precision), but it also precisely specifies the behavior of several basic operators, e.g. $+$, $\times$, $\sqrt{\ }$. In the rounding mode to the nearest, these operators shall return the *correctly-rounded* result, uniquely defined as the floating-point number closest to the exact mathematical value (in case of a tie, the number returned is the one with the even mantissa). The standard also defines three *directed rounding* modes (towards $+\infty$, towards $-\infty$, and towards 0) with similar correct rounding requirements on the operators.

The adoption and widespread use of this standard have increased the numerical quality and portability of floating-point code. It has improved confidence in such code and made it possible to construct proofs of numerical behavior [3]. Directed rounding modes are also the key to enable efficient *interval arithmetic* [4], a general technique to obtain validated numerical results.

This article is related to the IEEE-754 standard in two ways. Firstly, it discusses the issue of proving properties of numerical code, building upon the properties specified by this standard. Secondly, it addresses the implementation of elementary functions as recommended by the revision of the IEEE-754 standard: "they shall return results correctly rounded for the applicable rounding direction", so that their numerical quality matches that of the basic operators.

Elementary functions were left out of the original IEEE-754 standard in 1985 in part because the correct rounding property is much more difficult to guarantee for them than for the basic arithmetic operators. Specifically, the efficient implementation of a correctly rounded function [5] requires several evaluation steps, and for each step the designer needs to compute a bound on the overall evaluation error. Moreover, this bound should be tight, as a loose bound negatively impacts performance [6], [7].

This article describes a new approach to machine-checkable proofs of the *a priori* computation of such tight bounds. This approach is both interactive and easy to manage, yet much safer than a hand-written proof. It applies to error bounds as well as range bounds. Our approach is not restricted to the validation of elementary functions. It currently applies to any straight-line

- *F. de Dinechin is with LIP (ENS Lyon / CNRS / INRIA / UCBL / Université de Lyon).*
- *C. Lauter is with the Numerics team at Intel Corporation, Hillsboro, OR.*
- *G. Melquiond is with LRI (INRIA / Univ Paris-Sud 11 / CNRS).*

floating-point program of reasonable size (up to several hundreds of operations).

The novelty here is the use of a tool that transforms a high-level description of the proof into a machine-checkable version, in contrast to previous work by Harrison [8], [9] who directly described the proof of the implementation of some functions in all the low-level details. The Gappa approach is more concise and more flexible in the case of a subsequent change to the code. More importantly, it is accessible to people who do not belong to the formal proof community: this is the case of the two first authors of the present article.

This article is organized as follows. Section 2 surveys issues related to optimized floating-point programs, in particular elementary function implementations. Section 3 presents the Gappa tool, designed to address the challenge of proving tight bounds on ranges and errors in such programs. Section 4 discusses, in a tutorial manner, the proof of an elementary function using Gappa. The proof of a piece of code from the sine function of the `CRlibm` project[1] is build interactively as an extensive example.

2 Context of this work

2.1 Floating-point numbers are not real numbers

We have already mentioned that floating-point (FP) numbers do not possess basic properties of real numbers. Let us illustrate that with the Fast2Sum, a FP code sequence due to Dekker [10]:

```
s = a + b;
r = b - (s - a);
```

This sequence consists only of three operations. The first one computes the FP sum of the two numbers `a` and `b`. The second one would always return `b` and the third one 0, if this FP sum were exact. Because of the rounding, the sum is, however, often inexact. In IEEE-754 arithmetic with round-to-nearest, under certain conditions, this algorithm computes in `r` the error committed by this first rounding. In other words, it ensures that $\mathtt{r}+\mathtt{s} = \mathtt{a}+\mathtt{b}$ in addition to the fact that `s` is the FP number closest to $\mathtt{a}+\mathtt{b}$. The Fast2Sum algorithm provides us with an exact representation of the sum of two FP numbers as a pair of FP numbers, a very useful operation.

This example illustrates an important point, which pervades all of this article: FP numbers may be an approximation of the reals that fails to ensure basic properties of the reals, but they are also a very well-defined set of rational numbers, which have other well-defined properties, upon which it is possible to build mathematical proofs such as the proof of the Fast2Sum algorithm.

Let us come back to the condition under which the Fast2Sum algorithm works: the exponent of `a` should be larger than or equal to that of `b`, which is true for instance when $|\mathtt{a}| \geq |\mathtt{b}|$. In order to use this algorithm, one has first to prove that this property holds. Note that alternatives to the Fast2Sum exist for the case when one is unable to prove this condition. The version by Knuth [11] requires 6 operations instead of 3. Here, being able to prove the condition, which is a property on values of the code, will result in better performance.

The proof of the properties of the Fast2Sum sequence (three FP operations) requires several pages [10], and is indeed currently out of reach of the Gappa tool, basically because it cannot be reduced to manipulating ranges and errors. This is not a problem, since this algorithm has already been proven using formal proof systems [12]. We consider it as a generic building-block of larger floating-point programs, and the focus of our approach is to automate the proof of such larger programs. In the case of the Fast2Sum, this means proving the condition.

This work was initially motivated by a large class of such complex FP programs, implementations of elementary functions, which we now introduce in more details.

2.2 Floating-point elementary functions

Current floating-point implementations of elementary functions [13], [14], [15], [16], [17] have several features that make their proof challenging:

- The code size is too large to be safely proven by hand. In the first versions of the `CRlibm` project, the complete paper proof of a single function required tens of pages. It is difficult to trust such proofs.
- The code is optimized for performance, making extensive use of floating-point tricks such as the Fast2Sum above. As a consequence, classical tools of real analysis cannot be straightforwardly applied. Very often, considering the same operations on real numbers would simply be meaningless.
- The code is bound to evolve for optimization purpose, because better algorithms may be found, but also because the processor technology evolves. Such changes will require the proof to be rewritten, which is both tedious and error-prone.
- Much of the knowledge required to prove error bounds on the code is implicit or hidden, be it behind the semantics of the programming language (which defines implicit parenthesizing, for example), or in the various approximations made. Therefore, the mere translation of a piece of code into a set of mathematical variables that represent the values manipulated by this code is tedious and error-prone if done by hand.

Fortunately, implementations of FP elementary functions also have pleasant features that make their proof tractable:

- There is a clear and simple definition of the mathematical object that the floating-point code is supposed to approximate. This will not always be the case of e.g. scientific simulation code.
- The code size is small enough to be tractable, typically less than a hundred floating-point operations.

1. http://lipforge.ens-lyon.fr/www/crlibm/

- The control flow is simple, mostly consisting of straight-line code with a few tests but no loops.

The following elaborates on these features.

2.2.1 A primer on elementary function evaluation

Many methods exist for function evaluation [17]. Some are relevant only to fixed-point hardware evaluation, other to arbitrary-precision evaluation. We address here the evaluation of an elementary function for a fixed-precision floating-point format, typically for inclusion in a mathematical library (libm). It is classically [16], [17] performed by a polynomial approximation valid on a small interval only. A *range reduction* step brings the input number x into this small interval, and a *reconstruction* step builds the final result out of the results of both previous steps. For example, the logarithm may use as a range reduction the errorless decomposition $x = m \cdot 2^E$ of an FP number x into its mantissa m and exponent E. It may then evaluate the logarithm of the mantissa, and the reconstruction consists in evaluating $\log(x) = \log(m) + E \cdot \log(2)$. Note that current implementations typically involve several layered steps of range reduction and reconstruction. With current processor technology, efficient implementations [13], [14], [18] rely on large tables of precomputed values. See the books by Muller [17] or Markstein [19] for recent surveys on the subject.

In the previous logarithm example, the range reduction was exact, but the reconstruction involved a multiplication by the irrational $\log(2)$, and was therefore necessarily approximate. This is not always the case. For example, for trigonometric functions, the range reduction involves subtracting multiples of the irrational $\pi/2$, and will be inexact, whereas the reconstruction step consists in changing the sign depending on the quadrant, which is exact in floating-point arithmetic.

It should not come as a surprise that either range reduction or reconstruction are inexact. Indeed, FP numbers are rational numbers, but for most elementary functions, it can be proven that, with the exception of a few values, the image of a rational is irrational. Therefore, an implementation is bound, at some point, to manipulate numbers which are approximations of irrational numbers.

This introduces another issue which is especially relevant to elementary function implementation. One wants to obtain a double-precision FP result which is a good approximation to the mathematical result, the latter being an irrational most of the time. For this purpose, one needs to evaluate an approximation of this irrational to a precision better than that of the FP format.

2.2.2 Reaching better-than-double precision

Better-than-double precision is typically attained thanks to *double-extended* arithmetic on processors that support it in hardware. Otherwise, one may use *double-double* arithmetic, where a number is held as the unevaluated sum of two doubles, just as the 8-digit decimal number $3.8541942 \cdot 10^1$ may be represented by the unevaluated sum of two 4-digit numbers $3.854 \cdot 10^1 + 1.942 \cdot 10^{-3}$. Well-known and well-proven algorithms exist for manipulating double-double numbers [10], [11], the simplest of which is the Fast2Sum already introduced. These algorithms are costly, as each operation on double-double numbers requires several FP operations.

In this article, we consider implementations based on double-double arithmetic, because they are more challenging, but Gappa handles double-extended arithmetic equally well.

2.3 Approximation errors and rounding errors

The evaluation of any mathematical function entails two main sources of errors.

- Approximation errors (also called method errors), such as the error of approximating a function with a polynomial. One may have a mathematical bound for them (given by a Lagrange remainder bound on a Taylor formula for instance), or one may have to compute such a bound using numerics [20], [21], for example if the polynomial has been computed using Remez algorithm.
- Rounding errors, produced by most floating-point operations of the code.

The distinction between both types of errors is sometimes arbitrary. For example, the error due to rounding the polynomial coefficients to floating-point numbers is usually included in the approximation error of the polynomial. The same holds for the rounding of table values, which is accounted far more accurately as approximation error than as rounding error. This point is mentioned here because a lack of accuracy in the definition of the various errors involved in a given code may lead to one of them being forgotten.

2.4 Optimizations in floating-point code

Efficient code is especially difficult to analyze and prove because of all the techniques and tricks used by expert programmers.

For instance, many floating-point operations happen to be exact under some hypotheses, and the experienced developer of floating-point programs will arrange the code in such a way that these hypotheses are satisfied. Examples include multiplication by a power of two, subtraction of numbers of similar magnitude thanks to Sterbenz' Lemma [22], exact addition and exact multiplication algorithms (returning a double-double), multiplication of a small integer by a floating-point number whose mantissa ends with enough zeros, etc.

Expert programmers will also do their best to avoid computing more accurately than strictly needed. They will remove from the code some operations that are expected not to improve the accuracy of the result by much. This can be expressed as an additional approximation. However, it soon becomes difficult to know

what is an approximation to what, especially as the computations are reparenthesized to maximize floating-point accuracy.

To illustrate the resulting code obfuscation, let us introduce the piece of code that will serve as a running example along this article.

2.5 Example: a polynomial evaluation in double-double arithmetic

Listing 1 is an extract of the code of a sine function in the `CRlibm` library. These three lines compute the value of an odd polynomial $p(y) = y + s_3 \times y^3 + s_5 \times y^5 + s_7 \times y^7$ close to the Taylor approximation of the sine (its degree-1 coefficient is equal to 1). In our algorithm, the reduced argument y is ideally obtained by subtracting from the FP input x an integer multiple of $\pi/256$. As a consequence $y \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}]$.

However, as y is irrational (even transcendental), the implementation of this range reduction returns only an approximation of it. Due to the properties of sine, this approximation has to be more accurate than a double. Otherwise some information would be lost and the final result would be impossible to round correctly for some inputs. In our implementation, range reduction therefore returns a double-double $\mathtt{yh} + \mathtt{yl}$.

A modified Horner scheme is used for the polynomial evaluation:

$$p(y) = y + y^3 \times (s_3 + y^2 \times (s_5 + y^2 \times s_7)).$$

For a double-double input $y = \mathtt{yh} + \mathtt{yl}$, the expression to compute is thus

$$(\mathtt{yh}+\mathtt{yl})+(\mathtt{yh}+\mathtt{yl})^3\times(s_3+(\mathtt{yh}+\mathtt{yl})^2\times(s_5+(\mathtt{yh}+\mathtt{yl})^2\times s_7)).$$

The actual code uses an approximation of this expression. Indeed, the computation is accurate enough if all the Horner steps except the last one are computed in double-precision. Thus, y_l will be neglected for these iterations, and coefficients s_3 to s_7 will be stored as double-precision numbers written `s3`, `s5`, and `s7`. The previous expression becomes:

$$(\mathtt{yh} + \mathtt{yl}) + \mathtt{yh}^3 \times (\mathtt{s3} + \mathtt{yh}^2 \times (\mathtt{s5} + \mathtt{yh}^2 \times \mathtt{s7})).$$

However, if this expression is computed as parenthesized above, it has a poor accuracy. Specifically, the floating-point addition $\mathtt{yh}+\mathtt{yl}$ (by definition of a double-double) returns `yh`, so the information held by `yl` is completely lost. Fortunately, the other part of the Horner evaluation also has a much smaller magnitude than `yh` — this is deduced from $|y| \leq 2^{-7}$, therefore $|y^3| \leq 2^{-21}$. The following parenthesizing leads therefore to a much more accurate algorithm:

$$\mathtt{yh} + \left(\mathtt{yl} + \mathtt{yh} \times \mathtt{yh}^2 \times (\mathtt{s3} + \mathtt{yh}^2 \times (\mathtt{s5} + \mathtt{yh}^2 \times \mathtt{s7}))\right).$$

In this last version of the expression, only the leftmost addition has to be accurate. So we will use a Fast2Sum, which as we saw is an exact addition of two doubles returning a double-double stored in `sh` and `sl`. The other operations use the native — and therefore fast — double-precision arithmetic. We obtain the code of Listing 1.

Listing 1. Three lines of C

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

To sum up, this code implements the evaluation of a polynomial with many layers of approximation. For instance, variable `yh2` approximates y^2 through the following layers:

- y was approximated by $\mathtt{yh} + \mathtt{yl}$ with the relative accuracy $\varepsilon_{\text{argred}}$
- $\mathtt{yh} + \mathtt{yl}$ is approximated by `yh` in most of the computation,
- $\mathtt{yh}^2$ is approximated by `yh2`, with a floating-point rounding error.

In addition, the polynomial is an approximation to the sine function, with a relative error bound of $\varepsilon_{\text{approx}}$ which is supposed known (how it was obtained it is out of the scope of this paper [20], [21]).

Thus, the difficulty of evaluating a tight bound on an elementary function implementation is to combine all these errors without forgetting any of them, and without using overly pessimistic bounds when combining several sources of errors. The typical trade-off here will be that a tight bound requires considerably more work than a loose bound (and its proof might inspire considerably less confidence). Some readers may get an idea of this trade-off by relating each intermediate value with its error to confidence intervals, and propagating these errors using interval arithmetic. In many cases, a tighter error will be obtained by splitting confidence intervals into several cases, and treating them separately, at the expense of an explosion of the number of cases. This is one of the tasks that Gappa will helpfully automate.

2.6 Previous and related work

We have not yet explained why a tight error bound is required in order to obtain a correctly-rounded implementation. This question is surveyed in [7]. To sum it up, an error bound is needed to guarantee correct rounding, and the tighter the bound, the more efficient the implementation. A related problem is that of proving the behavior of *interval* elementary functions [18], [23]. In this case, a bound is required to ensure that the interval returned by the function contains the image of the input interval. A loose bound here means returning a larger interval than possible, and hence useless interval bloat. In both cases, the tighter the bound, the better the implementation.

As a summary, proofs written for versions of the `CRlibm` project up to version 0.8 are typically composed of several pages of paper proof and several pages of

supporting Maple script for a few lines of code. This provides an excellent documentation and helps maintaining the code, but experience has consistently shown that such proofs are extremely error-prone. Implementing the error computation in Maple was a first step towards the automation of this process; but although it helps to avoid computational mistakes, it does not prevent methodological mistakes. Gappa was designed, among other objectives, in order to fill this void.

There have been other attempts of assisted proofs of elementary functions or similar floating-point code. The pure formal proof approach by Harrison [8], [9], [24] goes deeper than the Gappa approach, as it accounts for approximation errors. However it is accessible only to experts of formal proofs, and fragile in case of a change to the code. The approach by Krämer *et al* [25], [26] relies on operator overloading and does not provide a formal proof.

3 The Gappa tool

Gappa[2] extends the interval arithmetic paradigm to the field of numerical code certification [27], [28]. Given the description of a logical property involving the bounds of mathematical expressions, the tool tries to prove the validity of this property. When the property contains unbounded expressions, the tool computes bounding ranges such that the property holds. For instance, the incomplete property "$x + 1 \in [2, 3] \Rightarrow x \in [?, ?]$" can be input to Gappa. The tool answers that $[1, 2]$ is a range of the expression x such that the whole property holds.

Once Gappa has reached the stage where it considers the property to be valid, it generates a formal proof that can be mechanically checked by an external proof checker. This proof is completely independent of Gappa and, more importantly, its validity does not depend on Gappa's own validity.

3.1 Floating-point considerations

Section 4 will give examples of Gappa's syntax and show that Gappa can be applied to mathematical expressions much more complex than just $x+1$, and in particular to floating-point approximations of elementary functions. This requires describing floating-point arithmetic expressions within Gappa.

Gappa only manipulates expressions on real numbers. In the property $x + 1 \in [2, 3]$, x is just a universally-quantified real number and the operator $+$ is the usual addition on real numbers $\mathbb{R}$. Floating-point arithmetic is expressed through the use of "rounding operators": functions from $\mathbb{R}$ to $\mathbb{R}$ that associate to a real number x its rounded value $\circ(x)$ in a specific format. These operators are sufficient to express properties of code relying on most floating-point or fixed-point arithmetics.

Verifying that a computed value is close to an ideal value can now be done by computing an enclosure of the error between these two values. For example, the property "$x \in [1, 2] \Rightarrow \circ(\circ(2 \times x) - 1) - (2 \times x - 1) \in [?, ?]$" expresses the absolute error caused during the floating-point computation of the following numerical code:

```
float x = ...;
assert(1 <= x && x <= 2);
float y = 2 * x - 1;
```

Infinities and NaNs (Not-a-Numbers) are not an implicit part of this formalism: the rounding operators return a real value and there is no upper bound on the magnitude of the floating-point numbers. This means that NaNs and overflows will not be generated nor propagated as they would in IEEE-754 arithmetic. However, one may still use Gappa to prove useful related properties. For instance, one can express in terms of intervals that overflows, or NaNs due to some division by 0, cannot occur in a given code. What one cannot prove are properties depending on the correct propagation of infinities and NaNs in the code.

3.2 Proving properties using intervals

Thanks to the inclusion property of interval arithmetic, if x is an element of $[0, 3]$ and y an element of $[1, 2]$, then $x + y$ is an element of the interval sum $[0, 3] + [1, 2] = [1, 5]$. This technique based on interval evaluation can be applied to any expression on real numbers. That is how Gappa computes the enclosures requested by the user.

Interval arithmetic is not restricted to this role though. Indeed the interval sum $[0, 3] + [1, 2] = [1, 5]$ do not only give bounds on $x + y$, it can also be seen as a proof of $x + y \in [1, 5]$. Such a computation can be formally included as an hypothesis of the theorem on the enclosure of the sum of two real numbers. This method is known as computational reflection [29] and allows for the proofs to be machine-checkable. That is how the formal proofs generated by Gappa can be checked independently without requiring any human interaction with a proof assistant.

Such "computable" theorems are available for the Coq [30] formal system. Previous work [31] on using interval arithmetic for proving numerical theorems has shown that a similar approach can be applied for the PVS [32] proof assistant. As long as a proof checker is able to do basic computations on integers, the theorems Gappa relies on could be provided. As a consequence, the output of Gappa can be targeted to a wide range of formal certification frameworks, if needed.

3.3 Other computable predicates

Enclosures are not the only useful predicates. As intervals are connected subsets of the real numbers, they are not powerful enough to prove some properties on discrete sets like floating-point numbers or integers. So Gappa handles other classes of predicates for an expression x:

$$\begin{aligned} \texttt{FIX}(x, e) &\equiv \exists m \in \mathbb{Z},\ x = m \cdot 2^e \\ \texttt{FLT}(x, p) &\equiv \exists m, e \in \mathbb{Z},\ x = m \cdot 2^e \wedge |m| < 2^p \end{aligned}$$

2. http://gappa.gforge.inria.fr/

As with intervals, Gappa can compute with these new predicates. For example,

$$\texttt{FIX}(x, e_x) \wedge \texttt{FIX}(y, e_y) \Rightarrow \texttt{FIX}(x + y, \min(e_x, e_y)).$$

These predicates are especially useful to detect real numbers exactly representable in a given format. In particular, Gappa uses them to find rounded operations that can safely be ignored because they do not contribute to the global rounding error. Let us consider the floating-point subtraction of two single-precision floating-point numbers $x \in [3.2, 3.3]$ and $y \in [1.4, 1.8]$. Note that Sterbenz' Lemma is not sufficient to prove that the subtraction is actually exact, as $\frac{3.3}{1.4} > 2$. Gappa is, however, able to automatically prove that $\circ(x-y)$ is equal to $x-y$.

As x and y are floating-point numbers, Gappa first proves that they can be represented with 24 bits each (assuming single precision arithmetic). As x is bigger than 3.2, it can then deduce it is a multiple of 2^{-22}, or $\texttt{FIX}(x, -22)$. Similarly, it proves $\texttt{FIX}(y, -23)$. The property $\texttt{FIX}(x-y, -23)$ then comes naturally. By computing with intervals, Gappa also proves that $|x-y|$ is bounded by 1.9. A consequence of these last two properties is $\texttt{FLT}(x-y, 24)$: only 24 bits are needed to represent $x-y$. So $x - y$ is representable by a single-precision floating-point number.

There are also some specialized predicates for enclosures. The following one expresses the range of an expression u with respect to another expression v:

$$\texttt{REL}(u, v, [a, b]) \equiv \exists \varepsilon \in [a, b],\ u = v \times (1 + \varepsilon).$$

This predicate is seemingly equivalent to the enclosure of the relative error $\frac{u-v}{v}$, but it simplifies proofs, as the error can now be manipulated even when v is potentially zero. For example, the relative rounding error of a floating-point addition vanishes on subnormal numbers (including zero) and is bounded elsewhere, so the following property holds when rounding to nearest in double precision:

$$\texttt{REL}(\circ(x + y), x + y, [-2^{-53}, 2^{-53}]).$$

3.4 Gappa's engine

Because basic interval evaluations do not keep track of correlations between expressions sharing the same terms, some computed ranges may be too wide to be useful. This is especially true when bounding errors, for example when bounding the absolute error $\circ(a) - b$ between an approximation $\circ(a)$ and an exact value b. The simplest option is to first compute the ranges of $\circ(a)$ and b separately and then subtract them. However, first rewriting the expression as $(\circ(a) - a) + (a - b)$ and then bounding $\circ(a) - a$ (a simple rounding error) and $a - b$ separately before adding their ranges usually gives a much tighter result. These rules are inspired by techniques developers usually apply by hand in order to certify their numerical applications.

Gappa includes a database of such rewriting rules and Section 3.5 shows how the user can expand this database with domain-specific hints. The tool applies these rules automatically, so that it can bound expressions along various evaluation paths. Since each of the resulting intervals encloses the initial expression, their intersection does, too. Gappa keeps track of the paths that lead to the tightest interval intersection and discards the others, so as to reduce the size of the final proof. It may happen that the resulting intersection is empty. This means that there is a contradiction between the hypotheses of the logical proposition, and Gappa can then deduce all the goals of the proposition from it.

Once the user enclosures have been proved (either by a direct proof or thanks to a contradiction), a formal proof is generated by retracing the paths that were followed when computing the ranges.

3.5 Hints

When Gappa is not able to satisfy the goal of the logical property, this does not necessarily mean that the property is false. It may just mean that Gappa does not have enough information about the expressions it tries to bound.

It is possible to help Gappa in these situations by providing *rewriting hints*. These are rewriting rules similar to those presented above in the case of rounding, but whose usefulness is specific to the problem at hand.

3.5.1 Explicit hints

A hint has the following form:

```
|Expr1 -> Expr2;
```

It is used to give the following information to Gappa: "I believe for some reason that, should you need to compute an interval for `Expr1`, you might get a tighter interval by trying the mathematically equivalent `Expr2`". This fuzzy formulation is better explained by considering the following examples.

1) The "some reason" in question will typically be that the programmer knows that expressions `A`, `B`, and `C`, are different approximations of the same quantity, and furthermore that `A` is an approximation to `B` which is an approximation to `C`. As previously, this means that these variables are correlated, and the adequate hint to give in this case is

   ```
   |A - C -> (A - B) + (B - C);
   ```

   It suggests to Gappa to first compute intervals for `A` − `B` and `B` − `C`, and then to sum them to get an interval enclosing `A` − `C`.
   As there are an infinite number of arbitrary `B` expressions that can be inserted in the right hand side expression, Gappa cannot try to apply every possible rewriting rule when it encounters `A` − `C`. So Gappa only tries a few `B` expressions, and the user has to provide the missing ones. Fortunately, as 3.5.2 will show, Gappa usually infers some useful `B` expressions, and it applies the corresponding

rewriting rules automatically. So an explicit hint is only needed in the most unusual cases.

2) Relative errors can be manipulated similarly. The hint to use in this case is

```
|(A-C)/C -> (A-B)/B + (B-C)/C +
  ((A-B)/B)*((B-C)/C)  {B <> 0, C <> 0};
```

This is still a mathematical identity, as one may check easily. The properties written between brackets tell Gappa when the rule is valid. Whenever Gappa wants to apply it, it has to prove the bracketed properties first.
As for the first rule, this second rule is needed quite often in a proof. So Gappa tries to infer some useful B expressions again, and it applies the rule automatically without any need for a user hint.

3) When x is an approximation of MX and a relative error $\varepsilon = \frac{\mathtt{x}-\mathtt{MX}}{\mathtt{MX}}$ is known to the tool, x can be rewritten $\mathtt{MX} \times (1+\varepsilon)$. This kind of hint is useful in combination with the following one.

4) When manipulating fractional terms such as $\frac{\mathtt{Expr1}}{\mathtt{Expr2}}$ where Expr1 and Expr2 are tightly correlated (for example one approximating the other), the interval division fails to give useful results if the intervals are wide. In this case, it is better to extract a correlated part A of the two expressions by stating $\mathtt{Expr1} = \mathtt{A} \times \mathtt{Expr3}$ and $\mathtt{Expr2} = \mathtt{A} \times \mathtt{Expr4}$. Hopefully, Expr3 and Expr4 are loosely (or even not) correlated, so the following hint gives a good enclosure of $\frac{\mathtt{Expr1}}{\mathtt{Expr2}}$.

```
|Expr1 / Expr2 -> Expr3 / Expr4 {A <> 0};
```

For a user-provided hint to be correct, a sufficient condition is: both sides of a rewriting rules are mathematically equivalent with respect to the field axioms of the real numbers. Gappa checks whether this is the case. If not, it warns the user that the hint should be reviewed for an error, for instance a mistyped variable name.

Note that, when an expression represents an error between two terms, e.g. $\mathtt{x} - \mathtt{Mx}$, the least accurate term should be written first and the most accurate one should be written last. The main reason is that the theorems of Gappa's database apply to expressions written in this order. This ordering convention prevents a combinatorial explosion on the number of paths to explore.

3.5.2 Automatic hints

After using Gappa to prove several elementary functions, it appeared that users kept writing the same hints, typically of the three first kinds enumerated in 3.5.1.

A new hint syntax, that is a kind of "meta-hint", was therefore introduced in Gappa:

```
|Expr1 ~ Expr2;
```

which reads "Expr1 approximates Expr2". This has the effect of automatically inserting rewriting hints for both absolute and relative differences involving Expr1 or Expr2. There may be useless hints among these inserted rewriting hints, but they are harmless.

Given this meta-hint, whenever Gappa encounters an expression of the form $\mathtt{Expr1} - \mathtt{Expr3}$ (for any expression Expr3), it applies the rule

$$\mathtt{Expr1} - \mathtt{Expr3} \rightarrow (\mathtt{Expr1} - \mathtt{Expr2}) + (\mathtt{Expr2} - \mathtt{Expr3}).$$

And when it encounters $\mathtt{Expr3} - \mathtt{Expr2}$, it tries to rewrite it as $(\mathtt{Expr3} - \mathtt{Expr1}) + (\mathtt{Expr1} - \mathtt{Expr2})$.

Note that, while meta-hints instruct Gappa to automatically insert rewriting hints, they are themselves automatically inserted by Gappa in a few cases. For instance, the user does not have to tell that Expr1 approximates Expr2, if Expr1 is the rounded value of Expr2. Gappa also inserts this meta-hint when an enclosure of the absolute or relative difference between Expr1 and Expr2 appears in the logical proposition that it is trying to prove.

3.5.3 Interval splitting and dichotomy hints

Finally, it is possible to instruct Gappa to split some intervals and perform its exploration on the resulting sub-intervals. There are several possibilities. For instance, the following hint

```
|$ z in (-1,2);
```

reads "Better enclosures may be obtained by separately considering the three cases $\mathtt{z} \leq -1$, $-1 \leq \mathtt{z} \leq 2$, and $2 \leq \mathtt{z}$."

Instead of being provided, the splitting points can also be automatically found by performing a dichotomy on the interval of z until the part of the goal corresponding to Expr has been satisfied for all the sub-intervals of z:

```
|Expr $ z;
```

3.5.4 Writing hints in an interactive way

Gappa has evolved to include more and more automatic hints, but most real-world proofs still require writing complex, problem-specific hints. Finding the right hint that Gappa needs could be quite complex and would require completely mastering its theorem database and the algorithms used by its engine. Fortunately, a much simpler way is to build the proof incrementally and question the tool by adding and removing intermediate goals to prove, as the extended example in next section will show. Before that, we first describe the outline of the methodology we use to prove elementary functions.

4 Proving elementary functions using Gappa

As in every proof work, style is important when working with Gappa: in a machine-checked proof, bad style will not in principle endanger the validity of the proof, but it may prevent its author from getting to the end. In the CRlibm framework, it may hinder acceptance of machine-checked proofs among new developers.

Gappa does not impose a style, and when we started using it there was no previous experience to get inspiration from. After a few months of use, we had improved

our "coding style" in Gappa, so that the proofs were much more concise and readable than the earlier ones. We had also set up a methodology that works well for elementary functions. This section is an attempt to describe this methodology and style.

The methodology consists in three steps, which correspond to the three sections of a Gappa input file.

- First, the C code is translated into Gappa equations, in a way that ensures that the Gappa proof will indeed prove some property of this program (and not of some other similar program). Then equations are added in order to describe what the program is supposed to implement. Usually, these equations are also in correspondence with the code.
- Then, the property to prove is added. It is usually in the form `hypotheses -> properties`, where the hypotheses are known bounds on the inputs, or contribution to the error determined outside Gappa, like the approximation errors.
- Finally, one has to add hints to help Gappa complete the proof. This last part is built incrementally.

The following sections detail these three steps.

4.1 Translating a floating-point program

We consider again the following C code, where we have added the constants:

```
s3 = -1.6666666666666665741e-01;
s5 =  8.3333333333333332177e-03;
s7 = -1.9841269841269841253e-04;
yh2 = yh * yh;
ts = yh2 * (s3 + yh2 * (s5 + yh2 * s7));
Fast2Sum(sh, sl,   yh, yl + yh * ts);
```

There is a lot of rounding operations in this code, so the first thing to do is to define Gappa rounding operators for the rounding modes used in the program. In our example, we use the following line to define `IEEEdouble` as a shortcut for IEEE-compliant rounding to the nearest double, which is the mode used in `CRlibm`.

```
@IEEEdouble = float<ieee_64,ne>;
```

Then, if the C code is itself sufficiently simple and clean, the translation step only consists in making explicit the rounding operations implicit in the C source code. To start with, the constants `s3`, `s5`, and `s7`, are given as decimal strings, and the C compilers we use convert them to (binary) double-precision FP numbers with round to nearest. We ensure that Gappa works with these same constants as the compiled C code by inserting explicit rounding operations:

```
s3 = IEEEdouble(-1.6666666666666665741e-01);
s5 = IEEEdouble( 8.3333333333333332177e-03);
s7 = IEEEdouble(-1.9841269841269841253e-04);
```

Then we have to do the same for all the roundings hidden behind C arithmetic operations. Adding by hand all the rounding operators, however, would be tedious and error-prone, and would make the Gappa syntax so different from the C syntax that it would degrade confidence and maintainability. Besides, one would have to apply without error the rules (well specified by the C99 standard [33]) governing for instance implicit parentheses in a C expression. For these reasons, Gappa has a syntax that instructs it to perform this task automatically. The following Gappa lines

```
yh2 IEEEdouble= yh * yh;
ts  IEEEdouble= yh2*(s3 + yh2*(s5 + yh2*s7));
```

define the same mathematical relation between their right-hand side and left-hand side as the corresponding lines of the C programs. This, of course, is only true under the following conditions:

- all the C variables are double-precision variables,
- the compiler/OS/processor combination used to process the C code respects the C99 and IEEE-754 standards and computes in double-precision arithmetic.

Finally, we have to express in Gappa the Fast2Sum algorithm. Where in C it is a macro or function call, for our purpose we prefer to ignore this complexity and simply express in Gappa the resulting behavior, which is a sum without error (again, we have here to trust an external proof of this behavior [10], [11]):

```
r IEEEdouble= yl + yh*ts;
s = yh + r;   # the Fast2Sum is exact, so s_h + s_l = y_h + r
```

Note that we are interested in the relative error of the sum `s` with respect to the exact sine, and for this purpose the fact that `s` has to be represented as a sum of two doubles in C is irrelevant.

More importantly, this adds another condition for this code translation to be faithful: As the Fast2Sum requires the exponent of `yh` to be larger than or equal to that of `r`, we now have to prove that. We delegate this process to Gappa by simply adding this precondition as a conclusion of the theorem to prove.

As a summary, for straight-line program segments with mostly double-precision variables, a set of corresponding Gappa definitions can be obtained by just replacing the C assignment operator with `IEEEdouble=` in the Gappa script, a straightforward and safe operation.

4.2 Defining ideal values

This code is supposed to evaluate the sine of its input. As a matter of fact, the property we intend to prove is a bound on the relative error of the computed value `s` with respect to this sine `SinY`. Gappa can be queried for this bound by typing `(s - SinY)/SinY in ?`. At this point, Gappa will not answer anything, since `SinY` has not been defined yet.

The expression `SinY` is the "mathematically ideal" value that the variable `s` tries to approximate. In order to prepare the proof, the mathematically ideal values of some other variables will also have to be defined. These values are the references with respect to which the errors are bounded. There is some choice in this notion

of mathematically ideal. For instance, what is the ideal value of yh2? It could be

- the exact square of yh (without rounding), or
- the exact square of yh + yl which yh approximates, or
- the exact square of the ideal reduced argument, which is usually irrational.

For our code, the mathematically ideal value for both yh+yl and yh will be the ideal reduced argument, which we write My. This is a real number defined as a function of the input variable x and the irrational π as detailed in Section 2.2.1. Similarly, the purest mathematical value that yh2 approximates is written My2 and will be defined as $\mathtt{My2} = \mathtt{My}^2$.

For the mathematical ideal of the polynomial approximation ts, we could choose, either the value of the function that the polynomial approximates, or the value of the same polynomial, but computed on My and without rounding error. Here we chose the latter, as it is syntactically closer to ts.

Here come a few naming conventions. The first one is obviously that the Gappa expressions representing the content of C variables have the same name. We also impose the convention that these names begin with a lowercase letter. In addition, Gappa variables for mathematically ideal terms will begin with a "M". The other intermediate Gappa variables should begin with capital letters to distinguish them from actual variables from the C code. Of course, related variables should have related and, wherever possible, explicit names. Again, these are conventions and are part of a proof style, not part of Gappa syntax. Indeed, the capitalization gives no information to the tool, and neither does the fact that expressions have related names.

For instance, it will be convenient to define a variable equal to yh + yl:

```
Yhl = yh + yl;
```

4.3 Defining what the code is supposed to compute

Defining mathematically ideal values amounts to defining in Gappa what the C code is supposed to implement. For instance, the line for ts can be seen as an approximated evaluation of a polynomial at point My:

```
My2 = My * My;
Mts = My2 * (s3 + My2 * (s5 + My2 * s7));
```

We have kept the polynomial coefficients in lower case: as already discussed in Section 2, the polynomial thus defined nevertheless belongs to the set of polynomial with real coefficients, and we have means to compute (outside Gappa) a bound of its relative error with respect to the function it approximates.

The link between ts and the polynomial approximating sine is also best expressed using mathematically ideal values:

```
PolySinY = My + My * Mts;
```

To sum up, PolySinY is the actual polynomial with the same coefficients s3 to s7 as in the C code, but evaluated without rounding error, and evaluated on the ideal value My that yh + yl approximates.

Another crucial question is: How do we define the real, ideal, mathematical function which we eventually approximate? Gappa has no builtin sine nor any other elementary function. The current approach can be described in English as: "$\sin(\mathtt{My})$ is a value which, as long as My is smaller than $6.3 \cdot 10^{-3}$, remains within a relative distance of $3.7 \cdot 10^{-24}$ of our ideal polynomial."

This relative distance between sine and the polynomial on this interval is computed outside Gappa. We used to depend on Maple's infinite norm, but it only returns an approximation, so this was in principle a weakness of the proof. We now use a safer, interval-based approach [20], [21], implemented with the Sollya tool.[3] This tool provides a theorem which could be expressed as

$$|\mathtt{My}| \leq 6.3 \cdot 10^{-3} \Rightarrow \left| \frac{\mathtt{PolySinY} - \mathtt{SinY}}{\mathtt{SinY}} \right| \leq 3.7 \cdot 10^{-24}$$

We inject this theorem in the Gappa script as an hypothesis of the property to prove:

```
   |My| <= 6.3e-03
/\ |(PolySinY - SinY)/SinY| <= 3.7e-24
/\ ... # (more hypotheses, see below)
-> (s - SinY) / SinY in ?
```

Concerning style, it makes the proof much clearer to add, from the beginning, as many definitions as possible for the various terms and errors involved in the computation:

```
# argument reduction error
Epsargred = (Yhl - My)/My;
# polynomial approximation error
Epsapprox = (PolySinY - SinY)/SinY;
# rounding errors in the polynomial evaluation
Epsround = (s - PolySinY)/PolySinY;
# total error
Epstotal = (s - SinY)/SinY;
```

4.4 Defining the property to prove

Thanks to the previous definition, the theorem to prove can be stated as follows:

```
{
 # Hypotheses
    |yl / yh| <= 1b-53
 /\ |My| <= 6.3e-03
 /\ |Epsargred| <= 2.53e-23
 /\ |Epsapprox| <= 3.7e-22

 # Goals to prove
 -> Epstotal in ?   # main goal of our theorem
 /\ |r/yh| <= 1     # validity of the Fast2Sum
}
```

The full initial Gappa script is given in Listing 2. It adds a more accurate definition of yh and yl, stating that they are double-precision numbers and that they form a disjoint double-double. Invoking Gappa on this script produces the following output:

3. http://sollya.gforge.inria.fr/

Listing 2. The initial Gappa file.

```
@IEEEdouble = float<ieee_64,ne>;

# y_h + y_l is a double-double (call it Y_hl)
yh = IEEEdouble(dummy1);
yl = IEEEdouble(dummy2);
Yhl = yh + yl;          # Below, there is also an hypothesis stating that |y_l| < ulp(y_h)

#———— Transcription of the C code ————————

s3 = IEEEdouble(-1.6666666666666665741e-01);
s5 = IEEEdouble( 8.3333333333333332177e-03);
s7 = IEEEdouble(-1.9841269841269841253e-04);
yh2 IEEEdouble=  yh * yh;
ts  IEEEdouble=  yh2 * (s3 + yh2*(s5 + yh2*s7));
r   IEEEdouble=  yl + yh*ts;
s              =  yh + r;     # no rounding, it is the Fast2Sum

#——— Mathematical definition of what we are approximating ———

My2 = My*My;
Mts = My2 * (s3 + My2*(s5 + My2*s7));
PolySinY = My + My*Mts;

Epsargred = (Yhl - My)/My;            # argument reduction error
Epsapprox = (PolySinY - SinY)/SinY;   # polynomial approximation error
Epsround = (s - PolySinY)/PolySinY;   # rounding errors in the polynomial evaluation
Epstotal = (s - SinY)/SinY;           # total error

#————————— The theorem to prove ————————————
{
 # Hypotheses
    |yl / yh| <= 1b-53
 /\ |My| <= 6.3e-03
 /\ |Epsargred| <= 2.53e-23
 /\ |Epsapprox| <= 2.26e-24

 # Goals to prove
 -> Epstotal in ?
 /\ |r/yh| <= 1
}
```

```
Results for |yl / yh| in [0, 1.11022e-16]
        and |My| in [0, 0.0063]
        and |Epsargred| in [0, 2.53e-23]
        and |Epsapprox| in [0, 2.26e-24]:
Warning: some enclosures were not satisfied.
Missing Epstotal /\ |r / yh|
```

4.5 With a little help from the user

This means that Gappa needs some help, in the form of hints. Where to start? There are several ways to interact with the tool to understand where it fails.

First of all, we may add additional goals to obtain enclosures for intermediate variables. For instance, when we add the goal "`|PolySinY| in ?`", Gappa answers "`|PolySinY| in [0, 0.0063]`". Gappa was able to deduce this enclosure from the enclosure of `My` (hypothesis) and from the syntax tree of `PolySinY`.

Similarly, we may notice that Gappa is unable to build enclosures of either `SinY` or `s`. This way it is possible to track the point where Gappa's engine gets lost, and provide hints to help it.

Among the values that Gappa cannot bound, there is `Yhl`. Yet Gappa knows some enclosures of both `My` and the relative error between `My` and `Yhl`. But Gappa is unable to formally prove a property by using the quotient $\frac{\texttt{Yhl}-\texttt{My}}{\texttt{My}}$ when `My` is equal to zero.

Therefore, in the early steps of a proof that involves relative errors, the `-Munconstrained` option of Gappa may be valuable. This mode weakens or removes the hypotheses of some theorems, so that the tool can go much further in its proof. Thanks to the option, the output of Gappa becomes:

```
Results for |yl / yh| in [0, 1.11022e-16]
        and |My| in [0, 0.0063]
        and |Epsargred| in [0, 2.53e-23]
        and |Epsapprox| in [0, 2.26e-24]:
|Yhl| in [0, 0.0063]
Warning: some enclosures were not satisfied.
Unproven assumptions:
  NZR(My)
```

It means that Gappa was able to bound `Yhl`, but the proof requires that the value `My` be nonzero, which Gappa is unable to prove given the current hypotheses. Fortunately, we can provide a positive lower bound on |`My`|. Not only does it ensure that `My` is nonzero, it will also ensure that relative errors are not arbitrarily big due to underflow. The bound was obtained thanks to the Kahan/Douglas algorithm [17], which uses the continued fractions of $\pi/256$ to find the floating-point

input that is the worst case of an additive argument reduction, namely the one that minimizes |My|.

We now try to obtain an enclosure of yh. For this purpose, we add two hints. The first one tells the tool that yh is almost equal to Yhl:

```
|yh ~ Yhl;
```

whereas the second one explains how to compute the relative error between the two of them given the ratio between yl and yh:

```
|(yh - Yhl) / Yhl -> 1 / (1 + yl / yh) - 1;
```

Gappa is now able to bound yh and ts, but not yl, and therefore neither r nor s. So we tell the tool how to deduce yl from yh.

```
|yl -> yh * (yl / yh)  { yh <> 0 };
```

At this point, the tool is able to bound all the variables of the C code and all their mathematically ideal values, assuming that some variables are not zero. Unfortunately, the relative errors between the variables and the ideal values are still out of reach.

For the following steps, we introduce two definitions in order to simplify the script:

```
|# just to make the hints lighter
|yhts = IEEEdouble(yh*ts);
|# remove last round on s
|S1 = yh + (yl + yhts);
```

Let us consider Epsround, which is the relative error between s and its mathematically ideal value PolySinY. Both values are sums: $\mathtt{yh} + \mathtt{r}$ and $\mathtt{My} + \mathtt{My} \cdot \mathtt{Mts}$. But Epsround should not be considered as the relative error of an addition, since r is not exactly an approximation of $\mathtt{My} \cdot \mathtt{Mts}$.

So we consider an intermediate expression $(\mathtt{yh} + \mathtt{yl}) + \mathtt{yhts}$ whose structure is closer to the structure of PolySinY. By reordering the terms of this expression, we get S1. So we tell Gappa that S1 is close to PolySinY and equal to the intermediate expression.

```
|S1 ~ PolySinY;
|S1 - (Yhl + yhts) -> 0;
```

Although they have the same structure, Gappa is still unable to bound the relative error between $(\mathtt{yh} + \mathtt{yl}) + \mathtt{yhts}$ and PolySinY. However, running Gappa in −Munconstrained mode produces a tight bound. So, this failure is only caused by some term being zero or a subnormal number and we will come back to it later.

A more pressing matter is that Gappa is unable to bound the relative error between $\mathtt{s} = \mathtt{yh} + \mathtt{r}$ et $\mathtt{S1} = \mathtt{yh} + (\mathtt{yl} + \mathtt{yhts})$. This is the relative error of an addition; querying Gappa about it shows that it successfully bounds the relative errors between its subterms. So the issue actually lies in the addition itself. In that case, we may help the tool by providing a rewriting hint for the quotient of the ideal subterms. There are two such quotients: $\mathtt{yh}/(\mathtt{yl}+\mathtt{yhts})$ and $(\mathtt{yl}+\mathtt{yhts})/\mathtt{yh}$. The second one looks much easier to manipulate, so we pass the following trivial hint to Gappa:

```
|(yl + yhts)/yh -> yl/yh + yhts/yh {yh <> 0};
```

Gappa is now able to prove all the properties in −Munconstrained mode, but it tells us that it had to assume that S1 is nonzero and that yh2, ts, and yhts are normal numbers. These properties seem true, especially due to the lower bound on |My|. Therefore we just have to suggest Gappa to consider positive and negative inputs separately:

```
|$ My in (0);
```

This hint is sufficient to finish the proof. It also has the effect that our third hint (the rewriting of yl) becomes useless and may be removed. The final Gappa script (Listing 3) is given at the end of this article, and is available from the distribution of CRlibm.

4.6 Summing up

Writing hints is the most time-consuming part of the proof, because it is the part where the designer's intelligence is required. However, we hope to have shown that it may be done very incrementally.

The example chosen in this article is actually quite complex: its Gappa proof consists of 63 lines, a sixth of which are hints. The bound found on Epstotal is $2^{-67.18}$ and is obtained in less than a second on a recent machine (the time can be longer when there is a dichotomy).

Some functions are simpler. We could write the proof of a logarithm implementation [7] with a few hints only [34]. One reason is that the logarithm never comes close to 0, so the full proof can be handled only with absolute errors, for which the load of writing hints is much lighter.

Once Gappa has proved all the goals, it can be asked for a formal proof script. In our case, the script is 2773-line long. It takes about one minute for the Coq proof assistant to mechanically check the correctness of the proof and hence of the presented algorithm. Note that the generated theorem contains as explicit hypotheses the three rewriting hints from Listing 3, since Gappa does not generate proofs for them.

5 Conclusion and perspectives

Validating tight error bounds on the low-level, optimized floating-point code typical of elementary functions has always been a challenge, as many sources of errors cumulate their effect. Gappa is a high-level proof assistant that is well suited to this kind of proofs.

Using Gappa, it is easy to translate a C function into a mathematical description of the operations involved with fair confidence that this translation is faithful. Expressing implicit mathematical knowledge one may have about the code and its context is also easy. Gappa uses interval arithmetic to manage the ranges and errors involved in numerical code. It handles most of the decorrelation problems automatically thanks to its builtin rewriting rules and an engine that explores the

possible rewriting of expressions to minimize the size of the intervals. All the steps of the proof are based on a library of theorems which allows Gappa to translate its computation process into a script mechanically checkable by a lower-level proof assistant such as Coq.

If Gappa misses some information for completing the proof, the user can add new rewriting rules in its Gappa script. The missing rules can be found by interrogating the tool while developing the script. Therefore, it is possible to quickly get a fully validated proof with good confidence that this proof indeed proves a property of the initial code. This process is, however, not instant. Writing a Gappa script requires exactly the same knowledge and cleverness a paper proof would. However, it requires much less work, since Gappa takes care of all the tedious details once the problem has been properly described (e.g. neglected terms, exact operations, reparenthesized expressions, and so on).

The current `CRlibm` distribution contains several bits of proofs using Gappa at several stages of its development. Although this development is not over, the current version of Gappa (0.12.3) is very stable and we safely consider generalizing the use of this tool in the future developments of `CRlibm`. The methodology and the proof style we presented in this paper are well suited to the validation of state-of-the-art elementary functions. This methodology is not set in stone, though. Indeed we may have to refine it further, as future functions may depend on unforeseen evaluation schemes with specialized proof processes.

Although we have insisted in this article on the interactivity of the tool, it can also successfully be used in blind mode. Current work targets code generators [35] that produce Gappa scripts along with the code. Typically, it is easy in such cases to first develop hints for one instance of the problem and then to generalize them to all the other instances.

REFERENCES

[1] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2009.

[2] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754, 2008.

[3] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–47, Mar. 1991.

[4] R. E. Moore, *Interval analysis*. Prentice Hall, 1966.

[5] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software*, vol. 17, no. 3, pp. 410–423, Sep. 1991.

[6] F. de Dinechin, A. Ershov, and N. Gast, "Towards the post-ultimate libm," in *17th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Jun. 2005, pp. 288–295.

[7] F. de Dinechin, C. Q. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *Theoretical Informatics and Applications*, vol. 41, pp. 85–102, 2007.

[8] J. Harrison, "Floating point verification in HOL light: the exponential function," University of Cambridge Computer Laboratory, Tech. Rep. 428, 1997.

[9] ——, "Formal verification of floating point trigonometric functions," in *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, ser. LNCS, vol. 1954. Springer, 2000, pp. 217–233.

[10] T. J. Dekker, "A floating point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[11] D. Knuth, *The Art of Computer Programming, vol.2: Seminumerical Algorithms*, 3rd ed. Addison Wesley, 1997.

[12] M. Daumas, L. Rideau, and L. Théry, "A generic library of floating-point numbers and its application to exact computing," in *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, ser. LNCS, vol. 2152. Springer, 2001, pp. 169–184.

[13] S. Gal, "Computing elementary functions: A new approach for achieving high accuracy and good performance," in *Accurate Scientific Computations*, ser. LNCS 235. Springer, 1986, pp. 1–16.

[14] P. T. P. Tang, "Table lookup algorithms for elementary functions and their error analysis," in *10th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Jun. 1991.

[15] S. Story and P. Tang, "New algorithms for improved transcendental functions on IA-64," in *14th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Apr. 1999, pp. 4—11.

[16] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas, "The libm library and floating-point arithmetic for HP-UX on Itanium," Hewlett-Packard company, Tech. Rep., april 2001.

[17] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser, 2006.

[18] D. Priest, "Fast table-driven algorithms for interval elementary functions," in *13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1997, pp. 168–174.

[19] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[20] S. Chevillard, C. Lauter, and M. Joldeş, "Certified and fast supremum norms of approximation errors," in *19th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, 2009, pp. 169–176.

[21] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter, "Efficient and accurate computation of upper bounds of approximation errors," LIP, CNRS/ENS Lyon/INRIA/Université de Lyon, INRIA, LORIA, CACAO project and Intel Corporation, Hillsboro, Oregon, Tech. Rep. RR 2010-2, Jan. 2010.

[22] P. H. Sterbenz, *Floating point computation*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[23] F. de Dinechin and S. Maidanov, "Software techniques for perfect elementary functions in floating-point interval arithmetic," in *7th Conference on Real Numbers and Computers*, Jul. 2006.

[24] J. Harrison, "Formal verification of square root algorithms," *Formal Methods in Systems Design*, vol. 22, pp. 143–153, 2003.

[25] W. Hofschuster and W. Krämer, "FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format," Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, Tech. Rep. Nr. 98/7, 1998.

[26] M. Lerch, G. Tischler, J. W. von Gudenberg, W. Hofschuster, and W. Krämer, "filib++ a fast interval library supporting containment computations," *Transactions on Mathematical Software*, vol. 32, no. 2, pp. 299–324, 2006.

[27] M. Daumas and G. Melquiond, "Generating formally certified bounds on values and round-off errors," in *6th Conference on Real Numbers and Computers*, 2004.

[28] G. Melquiond and S. Pion, "Formally certified floating-point filters for homogeneous geometric predicates," *Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 57–70, 2007.

[29] S. Boutin, "Using reflection to build efficient and certified decision procedures," in *Third International Symposium on Theoretical Aspects of Computer Software*, 1997, pp. 515–529.

[30] G. Huet, G. Kahn, and C. Paulin-Mohring, *The Coq proof assistant: a tutorial: version 8.0*, 2004. [Online]. Available: ftp://ftp.inria.fr/INRIA/coq/current/doc/Tutorial.pdf.gz

[31] M. Daumas, G. Melquiond, and C. Muñoz, "Guaranteed proofs using interval arithmetic," in *17th IEEE Symposium on Computer Arithmetic*. Cape Cod, Massachusetts, USA: IEEE Computer Society Press, 2005, pp. 188–195.

[32] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: a prototype verification system," in *11th International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.

[33] ISO/IEC, *International Standard ISO/IEC 9899:1999(E). Programming languages – C*, 1999.

Listing 3
The complete Gappa file.

```
@IEEEdouble = float<ieee_64,ne>;
# Convention 1: uncapitalized variables match the variables in the C code. Other variables begin with a capital letter.
# Convention 2: variables beginning with “M” are mathematical ideal.

# y_h + y_l is a double-double (call it Y_hl)

yh = IEEEdouble(dummy1);
yl = IEEEdouble(dummy2);
Yhl = yh + yl;  # There is also an hypothesis stating that |y_l| < ulp(y_h).

#———— Transcription of the C code ——————

s3 = IEEEdouble(-1.666666666666665741e-01);
s5 = IEEEdouble( 8.333333333333332177e-03);
s7 = IEEEdouble(-1.984126984126984125 3e-04);

yh2 IEEEdouble=  yh * yh;
ts  IEEEdouble=  yh2 * (s3 + yh2*(s5 + yh2*s7));
r   IEEEdouble=  yl + yh*ts;
s              =  yh + r;    # no rounding, it is the Fast2Sum

#——— Mathematical definition of what we are approximating ———

My2 = My*My;
Mts = My2 * (s3 + My2*(s5 + My2*s7));
PolySinY = My + My*Mts;

Epsargred = (Yhl - My)/My;            # argument reduction error
Epsapprox = (PolySinY - SinY)/SinY;   # polynomial approximation error
Epsround = (s - PolySinY)/PolySinY;   # rounding errors in the polynomial evaluation
Epstotal = (s - SinY)/SinY;           # total error

# Some definitions to simplify hints
yhts = IEEEdouble(yh*ts);             # just to make the hints lighter
S1 = yh + (yl + yhts);                # remove last round on s

#——————- The theorem to prove ——————
{
 # Hypotheses
    |yl / yh| <= 1b-53
 /\ |My| in [1b-200, 6.3e-03]  # lower bound guaranteed by Kahan-Douglas algorithm
 /\ |Epsargred| <= 2.53e-23
 /\ |Epsapprox| <= 3.7e-22
 /\ |SinY| in [1b-1000,1]

->

 # Goal to prove
    |Epstotal| <= 1b-67
 /\ |r/yh| <= 1
}

# ——————- Hints ———————————-

$ My in (0);

yh ~ Yhl;
(yh - Yhl) / Yhl -> 1 / (1 + yl / yh) - 1;

S1 ~ PolySinY;
S1 - (Yhl + yhts) -> 0;

(yl + yhts) / yh -> yl / yh + yhts / yh { yh <> 0 };
```

[34] F. de Dinechin, C. Q. Lauter, and G. Melquiond, "Assisted verification of elementary functions," LIP, Tech. Rep. RR2005-43, Sep. 2005.

[35] C. Q. Lauter and F. de Dinechin, "Optimising polynomials for floating-point implementation," in *8th Conference on Real Numbers and Computers*, Jul. 2008, pp. 7–16.

Florent de Dinechin was born in Charenton, France, in 1970. He received his DEA from the *École Normale Supérieure de Lyon* (ENS-Lyon) in 1993, and his PhD from *Université de Rennes 1* in 1997. After a postdoctoral position at Imperial College, London, he is now a permanent lecturer at ENS-Lyon in the *Laboratoire de l'Informatique du Parallélisme* (LIP). His research interests include computer arithmetic, software and hardware evaluation of elementary functions, computer architecture and FPGAs.

Christoph Lauter was born in Amberg, Germany, in 1980. He received a MSC degree in Computer Science from *Technische Universität München*, Germany, in 2005, and his PhD degree from *École Normale Supérieure de Lyon*, France, in 2008. He is member of the Numerics Group at *Intel Corporation*, Hillsboro, Oregon, USA. His research interests include correct rounding of elementary functions, computer arithmetic, computer algebra, numerical analysis and emulation of future hardware products.

Guillaume Melquiond was born in Nantes, France, in 1980. He received the MSC and PhD degrees in Computer Science from the *École Normale Supérieure de Lyon*, France, in 2003 and 2006. After a postdoctoral position at the Microsoft Research–INRIA joint center, he is now a researcher for the INRIA Saclay–Île-de-France in the ProVal team of the *Laboratoire de Recherche en Informatique* (LRI), Orsay, France. His research interests include floating-point arithmetic, formal methods for certifying numerical software, interval arithmetic, and C++ software engineering.