Assisted verification of elementary functions

Florent de Dinechin, Christoph Lauter, Guillaume Melquiond

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Assisted verification of elementary functions

Florent de Dinechin, Christoph Lauter, Guillaume Melquiond

N° 5683

Septembre 2005

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5683--FR+ENG

INRIA
RHÔNE-ALPES

Assisted verification of elementary functions

Florent de Dinechin, Christoph Lauter, Guillaume Melquiond

Thème NUM — Systèmes numériques
Projet Arénaire

Rapport de recherche n° 5683 — Septembre 2005 — 17 pages

Abstract: The implementation of a correctly rounded or interval elementary function needs to be proven carefully in the very last details. The proof requires a tight bound on the overall error of the implementation with respect to the mathematical function. Such work is function specific, concerns tens of lines of code for each function, and will usually be broken by the smallest change to the code (e.g. for maintenance or optimization purpose). Therefore, it is very tedious and error-prone if done by hand. This article discusses the use of the Gappa proof assistant in this context. Gappa has two main advantages over previous approaches: Its input format is very close to the actual C code to validate, and it automates error evaluation and propagation using interval arithmetic. Besides, it can be used to incrementally prove complex mathematical properties pertaining to the C code. Yet it does not require any specific knowledge about automatic theorem proving, and thus is accessible to a wider community. Moreover, Gappa may generate a formal proof of the results that can be checked independently by a lower-level proof assistant like Coq, hence providing an even higher confidence in the certification of the numerical code.

Key-words: proof assistant, floating-point, elementary functions, numerical code

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Vérification assistée de fonctions élémentaires

Résumé : L'implantation d'une fonction élémentaire avec arrondi correct ou d'une fonction élémentaire d'intervalle doit être prouvée dans les moindres détails. La preuve nécessite le calcul d'une borne fine sur l'erreur totale entre l'implémentation et la fonction mathématique exacte. Ce genre de travail est particulier à chaque fonction, concerne des dizaines de lignes de code, et risque d'être réduit à néant par une modification subséquente du code, pour raison de maintenance ou d'optimisation. Pour ces raisons, ce travail est très fastidieux et sujet à erreur s'il est effectué à la main. Cet article discute l'utilisation dans ce contexte de l'outil d'assistance à la preuve Gappa. Cet outil a deux avantages principaux sur les approches précédentes: sa syntaxe d'entrée est très proche de celle du code C à valider, et le calcul et la propagation des bornes d'erreurs sont automatisés autant que possible au moyen d'arithmétique d'intervalle. De plus, on peut l'utiliser de manière incrémentale pour prouver des propriétés complexes du code. Pourtant, l'outil ne nécessite pas de connaissances préalables en théorie de la démonstration, et est donc accessible à un public large. Enfin, Gappa produit une preuve formelle de ses résultats, preuve qui peut être vérifiée par des outils de preuve de plus bas niveau tels que Coq, pour encore plus de confiance dans la certification du code numérique.

Mots-clés : assistant de preuve, virgule flottante, fonctions élémentaires, code numérique

1 Introduction

The IEEE-754 standard for floating-point arithmetic [2] defines the usual floating-point formats (single and double precision) and precisely specifies the behavior of the basic operators $+, -, \times, \div$ and $\sqrt{\ }$ in four rounding modes (to the nearest, towards $+\infty$, towards $-\infty$ and towards 0). Its adoption and widespread use have increased the numerical quality and portability of floating-point code. It has improved confidence in such code and allowed construction of *proofs* of numerical behavior [11]. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) are also the key to enable efficient *interval arithmetic* [19].

However, the IEEE-754 standard does not specify as precisely the behavior of elementary functions (exp/log, sine/cosine, etc), so numerical portability is no longer assured as soon as the code calls such a function. More specifically, these functions do not always return the correctly rounded result (as do the basic operators). In a related way, interval elementary functions offered by mainstream interval packages do not return the tightest interval allowed by the floating-point representation.

The main reason for that is the Table Maker's Dilemma (TMD) [20]: given a floating-point number x and an elementary function f, computing the value of $\hat{y} = f(x)$ correctly rounded to a floating-point number $\circ(\hat{y})$ may require a very large intermediate accuracy, which is not known *a priori*. More specifically, a computer may evaluate an approximation y to the real number $\hat{y}$ with relative accuracy $\overline{\epsilon}$. This means that the real value $\hat{y}$ belongs to the interval $[y \cdot (1-\overline{\epsilon}), y \cdot (1+\overline{\epsilon})]$. Sometimes however, this information is not enough to decide correct rounding. For example, if $[y \cdot (1-\overline{\epsilon}), y \cdot (1+\overline{\epsilon})]$ contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is $\circ(\hat{y})$.

Ziv's technique [24] is to improve the accuracy $\overline{\epsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, the value of $f(x)$ is first evaluated using a quick approximation of accuracy $\overline{\epsilon}_1$. Knowing $\overline{\epsilon}_1$, it is possible to decide if rounding is possible, or if more accuracy is required, in which case the computation is restarted using a slower approximation of accuracy $\overline{\epsilon}_2$ greater than $\overline{\epsilon}_1$, and so on. This approach leads to good average performance, as the slower steps are rarely taken. Further studies [8] reduce the number of steps needed for correct rounding to two. The technique may be adapted to the computation of interval elementary functions as follows: in this case, knowing $\overline{\epsilon}_1$ allows to decides whether the rounded up or down value of $f(x)$ is known, or if one ulp *(unit in the last place)* should be added or removed to ensure the containment property.

The important notion here is that the correctness of the implementation (either of a correctly rounded function, or of an interval function) requires a bound on the overall error evaluating $f(x)$. Moreover, this bound should be tight: a loose bound means, for a correctly rounded function, launching the slower step more often than strictly needed, and hence degrading performance. For an implementation of an interval function, it means returning a larger interval result than possible, and hence useless interval bloat.

This article describes an approach to machine-checkable proofs of such tight error bounds that is both interactive and easy to manage, yet much safer than a hand-written proof. 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 [12] who directly described the proof of the implementation of an exponential function 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 outside the formal proof community.

This article is organized as follows. Next section describes in detail the challenges posed by automatic computation of tight error bounds. Section 3 describes the Gappa tool. Sections 4 and 5 give an overview on the techniques for proving an elementary function using Gappa and give an extensive example of the interactive construction of the proof.

2 Computing a tight error bound

The evaluation of an elementary function is classically 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 of x into its mantissa m and exponent E: $x = m \cdot 2^E$. It may then evaluate the logarithm of the mantissa, and the reconstruction consists in evaluating $\log(x) \approx \log(m) + E \cdot \log(2)$. Note that current implementations typically involve several layered steps of range reduction and reconstruction. With current processor technology, efficient implementations [10, 23, 22] rely on large tables of precomputed values. See the books by Muller [20] or Markstein [17] for recent surveys on the subject.

Correctly rounded functions, but also most recent implementations which follow the current quality standard of a maximum error bound only slightly larger than the half-ulp, require to perform some of these computations in a precision larger than the target precision [10]. This is obtained thanks to *double-extended* arithmetic on processors that support it, or *double-double* arithmetic, where a number is held as the unevaluated sum of two doubles. Well-known and well-proven algorithms exist for manipulating double-double numbers [9, 15]. In this article, we will consider implementations based on double-double arithmetic, because they are more challenging, but Gappa handles double-extended arithmetic equally well.

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.

More generally, the evaluation of an elementary function using such algorithms entails two main sources of errors.

- Approximation errors (also called methodical errors), such as the error of approximating a function with a polynomial. One may have a mathematical bound for them (given by a Taylor formula for instance), or one may have to compute such a bound using numerics (if the polynomial has been computed using Remez algorithm in particular).

- 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 evaluated more accurately if counted as approximation error than if counted as rounding error. This point is mentioned here because a lack of accurateness in the definition of the various errors involved in a given code may lead to forgetting to take one of them into account.

Note also that many floating-point operations are exact, and the experienced author of floating-point code will try to use them. Examples include multiplication by a power of two, subtraction of numbers of similar magnitude thanks to Sterbenz' Lemma, 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 zeroes, etc.

On the other hand, an optimized elementary function implementation will stack approximation over approximation to avoid computing more accurately than strictly needed. A good example is the case of trigonometric functions. Their (inexact) range reduction has to return a double-double (otherwise there is no hope of achieving less than one ulp accuracy). However, a subsequent polynomial evaluation using the Horner scheme should not be computed fully in expensive double-double arithmetic. The less significant double of the reduced argument is therefore neglected in the first Horner steps. Thus, the argument to these Horner steps is a double that already cumulates two approximations: that of the inexact range reduction, and that of dropping the lower double.

This example shows that it takes considerable discipline to define properly what is the error of what with respect to what.

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 considerable 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.

As a summary, proofs written for version of the `crlibm` project up to version 0.8 [1] are typically composed of several pages of paper proof and several pages of supporting Maple 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 if it helps avoiding computation mistakes, it does not prevent methodological mistakes. Gappa was designed in order to fill this void.

3 The Gappa tool

Gappa[1] is a tool that extends the interval arithmetic paradigm to the field of numerical code certification [5, 18]. Given the description of a logical property involving the bounds of mathematical expressions, the tool will try to prove the validity of this property. When the property contains unbounded expressions, the tool will compute bounding ranges such that the property holds. Once Gappa has reached the stage where it considers the property to be valid, it will generate a formal proof that can be checked by an independent proof assistant.

This incomplete property "$x + 1 \in [2, 3] \Rightarrow x \in [?, ?]$" could be fed to Gappa. The tool would answer that $[1, 2]$ is a range of the expression x such that the whole property holds. Gappa would also output a formal proof that states the property. This proof is completely independent of Gappa and 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 more complex than just $x+1$. In particular it can be used to certify the accuracy of the floating-point approximation of elementary functions. This requires feeding Gappa with floating-point arithmetic expressions.

Indeed, in the expression $x+1$, x is just a universally quantified real number, and the operator $+$ is the addition on the real numbers $\mathbb{R}$. Gappa only manipulates real expressions, floating-point arithmetic is expressed through "rounding operators": functions from $\mathbb{R}$ to $\mathbb{R}$ that associate to a real number x its rounded value $\circ(x)$.

Thanks to this formalism, floating-point rounding errors can now be evaluated through the logical properties and mathematical expressions Gappa knows how to handle. For example, the property "$\circ(x) \in [1, 2] \Rightarrow \circ(\circ(x) - 1) - (\circ(x) - 1) \in [?, ?]$" will give a bound on the absolute error caused during the floating-point computation of the following numerical code:

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

It can be noted that Gappa is perfectly aware of lemmas like Sterbenz'. As a consequence, it will notice that there is no rounding error in this particular case: the singleton range $[0, 0]$

[1] http://lipforge.ens-lyon.fr/www/gappa/

is enough for the property to hold. Without this lemma, Gappa would have generated a proof involving the range $[-2^{-24}, 2^{-24}]$ instead, for a rounding to nearest in IEEE-754 single precision.

Infinities and 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 use Gappa to prove very useful properties, for instance that overflows, or NaNs due to some division by 0, cannot occur in a given code: This can be expressed in terms of intervals. What one cannot prove are properties depending on the 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 reflexivity [3] 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 [14] and HOL Light [13] proof assistants. Previous work [6] on using interval arithmetic for proving numerical theorems has shown that a similar approach can be applied for the PVS [21] 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 Gappa's theorem library

Gappa relies on a database of such theorems. They can be split in three main categories. First are the theorems describing the properties of the arithmetic on real numbers. Then come the theorems on rounding operations. And finally there is a set of rewriting rules to increase the interval correlation between the terms of an expression.

3.3.1 Basic arithmetic on reals

There are more than thirty theorems on the basic arithmetic operators ($+$, $-$, $\times$, and so on). These theorems are applied through computations on intervals. The bounds are dyadic numbers $m \cdot 2^n$ and the computations are done with the Boost interval arithmetic library [4] and MPFR[2]. Gappa keeps a trace of these computations so that it can generate the final proof.

3.3.2 Rounding

There are also various theorems about the tens of rounding operators Gappa knows about. Among which are the four rounding modes for the usual precisions of IEEE-754. With the exception of infinity and NaN generation and propagation, they reflect the IEEE-754 standard closely. In particular, they take subnormal numbers into account.

For a given rounding operator $\circ$, Gappa usually needs to know how to compute the enclosures of the rounded expression $\circ(a)$, of the rounding absolute error $\circ(a) - a$, and of the rounding relative error $\frac{\circ(a)-a}{a}$. These enclosures may depend on the enclosures of a or $\circ(a)$. There are even theorems computing the enclosure of $\circ(a)$ out of the enclosure of $\circ(a)$. For example, consider the rounding

[2] http://www.mpfr.org/

of a real number to an integer: if $\lfloor x \rceil$ is in the interval $[1.3, 2.7]$, then it means that $\lfloor x \rceil$ is equal to 2.

3.3.3 Expression rewriting

Interval computations are unfortunately plagued by decorrelation. For example, when bounding the absolute error $\circ(a) - b$ between an approximation $\circ(a)$ and an exact value b, first computing the enclosures of $\circ(a)$ and b separately and then subtracting them will lead to a result so wide that it is useless. However, first rewriting the expression as $(\circ(a) - a) + (a - b)$ and then computing $\circ(a) - a$ (a simple rounding error) and $a - b$ separately before subtracting them will usually give a much tighter result. That is why Gappa also has about thirty theorems on expression rewriting in its database.

Thanks to them, a single expression can be bounded along various evaluation paths. Since any of these resulting intervals encloses the initial expression, their intersection does, too. Gappa will keep the paths that leads to the tightest interval intersection and will discard the others so as to reduce the size of the final proof. It may happen that the resulting intersection is empty; it means that there is a contradiction between the hypotheses of the logical property and Gappa will use it to prove all the goals of the logical property.

3.4 Gappa's engine

Gappa will first extract the numerical intervals from the hypotheses of the logical property and analyze which intermediate expressions it may be helpful to bound in order to finally get to the goals of the logical property. Once this set of expressions is computed, the tool will progressively try to evaluate the range of all of them, until none of its theorem can help it anymore to further reduce their enclosures. At this time, if the goals of the logical property are satisfied by these intervals, Gappa will have succeeded in verifying the logical property and will generate the corresponding formal proof.

Otherwise, Gappa was unable to prove the property. It does not necessarily mean that the property is false, it may just mean that Gappa had not enough information about the property. The next section will explain how to help Gappa in these situations by providing some rewriting hints.

Remark that more theorems may be added in the future to the theorem database, to reflect e.g. a floating-point trick that has never been used so far. This may not break existing proof, at worst slow down the exploration and refine some of the computed intervals.

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 to get 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. In 6 months of use, we have improved our "coding style" in Gappa, so that current proofs are much more concise and readable than the earlier ones. We have also set up a methodology that works well for elementary functions. This section is an attempt to describe this methodology and style. We are aware that they may be inadequate for other applications, and that even for elementary functions they could be improved further.

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 pro-

gram). Then equations are added describing 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* which indicate to the Gappa engine how to unroll the proof, or make explicit the implicit knowledge one has about the code. This last part is built incrementally.

The following details these three steps.

4.1 Notation conventions

Before starting, one has to remember that for Gappa there is only one type of variables, which may hold arbitrary intervals. In the proof of an elementary function, to help managing the problem of "what is the error of what with respect to what" evoked in Section 2, we use the following conventions: Gappa variables behaving exactly like the C variables have exactly the same name, which should begin with a lower case letter. Variables for mathematically ideal terms begin with a "`M`". All the other intermediate variables will begin with capital letters. In addition, related variables should have related and, wherever possible, explicit names.

These conventions are best explained with an example: Consider the following code bit, extracted from the proof of Section 5.

```
1 Mul12(&zhSquareh, &zhSquarel, zh, zh);
2 zhCube = zh * zhSquareh;
```

It inputs a variable `zh`, computes exactly its square as a double-double `zhSquareh` + `zhSquarel` using Dekker's algorithm (here implemented as a call to the `Mul12` function), then computes an approximation to its cube. We will need to analyse this code at least one variable `MZCube`, which is the mathematical value that `zhCube` intends to approximate in this code, and one variable `MZSquare`.

Now the notion of "mathematically ideal" may be quite subtle. In our example, `zh` is itself an approximation to an ideally reduced argument, noted of course `MZ`. Therefore, it should be clear that the equation defining `MZSquare` is `MZSquare = MZ * MZ;` and not `MZSquare = zh * zh;`: Although the squaring was exact in the code, it did not compute the square of the exact reduced value.

Again, these are conventions and are part of our proof style, not part of Gappa syntax: the capitalization will give no information to the tool, and neither will the fact that variables have related names.

Another useful convention will be to define variables for absolute and relative errors beginning respectively with `delta` and `epsilon`, as in the following example:

```
deltaZh = zh - MZ;
epsilonZhCube = (zhCube - MZCube) / MZCube;
```

This last convention makes the proofs much readable and eases the task of writing hints, especially when dealing with relative errors.

So far, we failed to converge on conventions that would be both helpful and concise for the full names of these errors, and for managing double-double arithmetic.

4.2 Translating a FP program

If the C code is itself sufficiently simple and clean, this step only consists in making explicit the rounding operations implicit in the C source code. The syntax `<float64ne>(Expr)` correspond

to a rounding to the nearest double of Expr, so for instance `<float64ne>(a+b)` mimicks the IEEE-754-compliant addition with correct rounding.

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) governing implicit parentheses in a C expression. For these reasons, Gappa has a syntax that instructs it to perform this task automatically, illustrated by the following example: The C line

```
q = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
```

and the Gappa line

```
q <float64ne>= c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
```

define the same mathematical relation between their right-hand side and left-hand side, under the conditions that all the C variables are double-precision variables, that the Gappa variables on the right-hand side mimick them (see the lowercase convention), and also, of course, that the compiler/OS/processor combination used to process the C code respects the C99 and IEEE-754 standards and computes in double-precision arithmetic.

All this means that for straight-line program segments with mostly double-precision variables, a set of corresponding Gappa definitions can be obtained straightforwardly by just replacing the C `=` with Gappa `<float64ne>=`, a very safe operation.

There is another syntax which helps expressing properties of double-double operators, for which a bound on the relative error is known since Dekker [9, 16], but proven outside Gappa. `X = <relative,103>(Expr);` states that $\left|\frac{\mathtt{X}-\mathrm{Expr}}{\mathrm{Expr}}\right| \leq 2^{-103}$. Using this syntax, the C line

```
Add22(&xh,&xl,  ah,al, bh,bl);
```

which expresses the double-double operation $X = A + B$, may be translated to the line

```
X = <relative,103>(A + B);
```

This does not, however, define exactly the same mathematical relation: the Gappa version does not imply, for instance, that `X` can be written exactly as the sum of two doubles `xh` and `xl`. This has to be expressed explicitly if needed.

4.3 Defining ideal values

The next operation to carry out is to define in Gappa what the C code is supposed to implement. For instance, using our previous conventions, the line for `q` was probably evaluating the value of the same polynomial of the ideal `MZ`:

```
MQ = c3 + MZ * (c4 + MZ * (c5 + MZ * (c6 + MZ * c7)));
```

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 know how to compute in Maple a bound of its relative error with respect to the function it approximates.

Another approach could be to use a Taylor polynomial, in which case the approximation error would be given by the rest in the Taylor formula, the ideal polynomial would be the Taylor one, it would have ideal Taylor coefficients (beginning with M), some of which would have to be rounded to FP numbers to appear in the program (lowercase). Gappa could handle it, too.

Another question is, how do we define the mathematical function? Gappa has no builtin sine or logarithm. The current approach can be described in English as: "$\log(1 + Z)$ is a value which, if Z is smaller 2^{-8}, is within a relative distance of 2^{-63} of our ideal polynomial". In Gappa, this translates to hypotheses in the property to prove (with the Gappa syntax `1b-8` for 2^{-8}):

```
Z in [-1b-8,1b-8] /\
(MP - MLog1pZ) / MLog1pZ  in [-1b-63, 1b-63] /\ ...
```

Here the interval of Z is defined by the range reduction, and the 2^{-63} bound has to be computed outside Gappa (for instance thanks to an infinite norm evaluated in Maple). This is in principle a weakness of the proof, however we take some safety margins, and on the considered intervals, elementary functions are regular enough to trust Maple's infinite norm.

Then we may use the reconstruction associated with the argument reduction used to define the mathematical function on the whole of its interval, as for the logarithm:

```
MLogx = MLog1pZ + E * MLog2;
```

4.4 Defining the property to prove

The theorem to prove is expressed as implications using classical first-order logic, with some restrictions. In practice we usually list a conjunction of hypotheses, the `->` operator, and a conjunction of conclusions to prove. For a full example, see next section.

4.5 Hints

The hint part reflects the work humans still must do in order to prove the numerical properties of the code.

The hints have the following form:

```
Expr1 -> Expr2;
```

which 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 variables `xh`, `MX` and `X` are different approximations of the same quantity, and furthermore that `xh` is an approximation to `X` which is an approximation to `MX`. Suppose that at some point Gappa has to compute `xh - MX`, and even that it already has a good interval for `xh` and a good interval for `MX` (the values will be quite similar since `xh` approximates `MX`). In this case, standard interval arithmetic will lead to a very coarse interval for `xh - MX`.

   The adequate hint to give in this case is

   ```
   xh - MX -> (xh - X) + (X - MX);
   ```

   It will instruct Gappa to first compute intervals for `xh - X` and `X - MX` (both of which will be small) and sum them to get an interval for `xh - MX` (which will thus be tight as well).

   Note that if one defines `delta*` intermediate variables for absolute errors, this hint will be equivalently written:

   ```
   delta -> delta1 + delta2;
   ```

2. Relative errors can be manipulated similarly. Given $\epsilon = \frac{x-MX}{MX}$, $\epsilon_1 = \frac{x-X}{X}$, and $\epsilon_2 = \frac{X-MX}{MX}$, the next hint may be used. In particular it is needed for the integration of polynomial approximation errors into the final error estimate.

   ```
   epsilon -> epsilon1 + epsilon2
              + epsilon1 * epsilon2;
   ```

   This is still a mathematical identity as one may check easily by developing the definitions.

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

4. When manipulating fractional terms such as $\frac{\texttt{Expr1}}{\texttt{Expr2}}$ where `Expr1` and `Expr2` are correlated (for example one approximating the other), the interval division fails to give useful results if the interval for `Expr2` comes close to 0. In this case, one will try to write Expr1 = $\texttt{A} \cdot$ Expr3 and Expr2 = $\texttt{A} \cdot$ Expr4, so that the interval on Expr4 does not come close to 0 anymore. The following hint is then appropriate:

```
Expr1 / Expr2 -> Expr3 / Expr4;
```

   This rewriting rule is only valid if `A` is not zero, so the case $\texttt{A} = 0$ has to be handled separately.

All these hints are correct if both sides are mathematically equivalent. Gappa therefore checks this automatically. If the test fails - which is rare - it emits a warning to the user that he or she must review the hint by hand. Therefore, writing even complex hints is very safe: one may not introduce an error in the proof by writing hints which do not emit warnings.

However, finding the right hint that Gappa needs could be quite complex without 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 the following section shows.

5 Extended example: a logarithm

This section computes a relative error bound on the evaluation to a double-double of a polynomial approximating $\log(1+Z)$ where Z is a reduced argument. This computation is the core of the first step in `crlibm`'s current portable implementation of the natural logarithm. The argument reduction used is errorless, but the reduced argument needs to be stored on a double-double, so $Z = z_h + z_l$. The proof that the argument reduction is exact is done by hand, and the reconstruction introduces no new difficulty (it merely consists in two successive double-double additions), so we do not show it here for the sake of brevity. For a full description of this implementation, including the second step, see [7].

5.1 Algorithm

This polynomial evaluation inputs the double-double $Z = z_h + z_l$, and should return a double-double approximation to $p(Z)$. Evaluating the whole of the polynomial using double-double arithmetic would not be efficient: instead, the polynomial approximating $\log(1+Z)$ is written as follows:

$$p(Z) = Z - \frac{1}{2} \cdot Z^2 + Z^3 \cdot q(Z) \tag{1}$$

and the respective terms are evaluated as follows:

- $\frac{1}{2} \cdot Z^2 = \frac{1}{2} \cdot (z_h + z_l)^2$ is approximated by $\frac{1}{2} \cdot z_h^2 + z_h z_l$, where z_h^2 is computed exactly as a double-double.
- Z^3 is approximated by z_h^3 computed in double precision,
- $q(Z)$ is a polynomial with double-precision coefficients, and is approximated by $q(z_h)$ so that it can be evaluated entirely in double.

The corresponding C code is given below.

```
1 q = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
2 Mul12(&zhSquareh, &zhSquarel, zh, zh);
3 zhCube = zh * zhSquareh;
4 polyUpper = zhCube * q;
```

```
5 zhSquareHalfh = zhSquareh * -0.5;
6 zhSquareHalfl = zhSquarel * -0.5;
7 zhzl = -1 * (zh * zl);
8 Add12(t1h, t1l, polyUpper, zhzl);
9 Add22(&t2h, &t2l, zh, zl, zhSquareHalfh, zhSquareHalfl);
10 Add22(&ph, &pl, t2h, t2l, t1h, t1l);
```

Here `Add12` (also known as Fast2Sum) is a sequence computing the exact sum of two doubles as a double-double. Similarly, `Mul12` computes the exact product of two doubles as a double-double. Finally, the procedure called `Add22` computes as a double-double the sum of two double-double numbers with a relative error less than 2^{-103} [9, 16].

The code is a typical example of floating-point code written for a target accuracy of about 2^{-62} (neglecting the lower significant argument z_l in all terms where it is not strictly needed, for instance). It also expresses some parallelism (line 1 and lines 2-3 can be evaluated concurrently). We now establish the proof of the error of this code step by step, in the typical way one would be using Gappa in order to incrementally prove the implementation of any elementary function.

5.2 Translating the C code

Following the guidelines of Section 4, the C code can be translated into Gappa syntax as follows:

```
1 zh = <float64ne>(Z);
2 zl = Z - zh;
3 q <float64ne>=  c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
4 ZhSquarehl = zh * zh;
5 zhSquareh = <float64ne>(ZhSquarehl);
6 zhCube = <float64ne>(zh * zhSquareh);
7 polyUpper <float64ne>= zhCube * q ;
8 ZhSquareHalfhl = -0.5 * ZhSquarehl;
9 zhzl = -1 * <float64ne>(zh * zl);
10 T1hl = polyUpper + zhzl;
11 T2hl = <relative,103>(Z + ZhSquareHalfhl);
12 Phl = <relative,103>(T2hl + T1hl);
```

Note that there is no rounding operator at line 8: we know that a multiplication by -0.5 is exact in IEEE-754-compliant arithmetic for the range of values we consider. The same holds for the multiplication by -1, line 9.

5.3 Expressing the mathematical property expected from the code

The next step is to express in Gappa what this code is intended to compute. The "mathematically ideal" definition of the polynomial is expressed in the form given by Eq. (1), since it is the form that the code uses. Additionally, we can define ϵ, which is the error to be evaluated, and `epsilonApproxPoly` which is actually there to define (as per Section 4.3) the ideal mathematical value of $\log(1+Z)$, noted `MLog1pZ`:

```
14 MQ = c3 + Z * (c4 + Z * (c5 + Z * (c6 + Z * c7)));
15 MZSquare = Z * Z;
16 MZCube = Z * MZSquare;
17
18 MP = Z - 0.5*MZSquare + MZCube*MQ;
19
20 epsilon = (Phl - MLog1pZ) / MLog1pZ;
21 epsilonApproxPoly = (MP - MLog1pZ) / MLog1pZ;
```

Note that there is no need for an `MZ` variable here, because argument reduction is exact (we would have `Z` = `MZ`).

Then we express the theorem to prove. The hypotheses on the input $Z = z_h + z_l$ are proven in the paper proof of the argument reduction: Z is bounded by $|Z| \leq 2^{-8}$, and cannot be less than 2^{-200} in magnitude if it is not exactly 0. This knowledge is expressed in Gappa as follows:

```
|Z| in [1b-200,1b-8] /\ |zl| in [1b-300,1]
```

(the Gappa syntax `1b-200` stands for 2^{-200}). Another hypothesis is the polynomial approximation error, also computed outside Gappa. Therefore the theorem to prove is written as follows.

```
{   |Z| in [1b-200,1b-8]
 /\ |zl| in [1b-300,1]
 /\ epsilonApproxPoly in [-1b-63,1b-63]
  -> epsilon in [-1b-62,1b-62]  }
```

5.4 From the maths to the code

Then we need some additional definitions to reflect the transition from the mathematical equations to the code. When the code closely matches the mathematics, this is easy:

```
MPolyUpper = MQ * MZCube;
MZSquareHalf = -0.5 * MZSquare;
MZhzl = -1 * (zh * zl);
```

However, for optimization purposes, clever code may reorder the terms of the pure mathematical equations, neglect terms, etc. As an example, our code does not implement exactly equation (1), but the following equivalent one:

$$\begin{aligned} p(Z) &= Z - \frac{1}{2}\cdot Z^2 + Z^3\cdot q(Z) \\ &= Z - \frac{1}{2}\cdot (z_h+z_l)^2 + Z^3\cdot q(Z) \\ &= \left(Z^3\cdot q(Z) - z_h z_l\right) + \left(Z - \frac{1}{2}\cdot z_h^2\right) \quad - \frac{1}{2}\cdot z_l^2 \end{aligned}$$

The code implements the last line and neglects the smaller term $\frac{1}{2}\cdot z_l^2$.

We want to define mathematically ideal values for the corresponding `T1hl` and `T2hl`, such that `MT1` + `MT2` = `MP` to reflect the last `Add22`. For this to be true, the neglected $\frac{1}{2}\cdot z_l^2$ must be incorporated to either `MT1` or `MT2`. We chose `MT2` because its magnitude is higher:

```
MT1 = MPolyUpper + MZhzl;
MT2 = Z + (MZSquareHalf - MZhzl);
```

Contrary to the previous steps, this step is not obvious, mostly because it reflects something clever in the code.

Having done that, we have mathematical ideals for all the variables in the code. This will now allow us to express `epsilon`s for all these variables as the proof progresses.

5.5 Analysing and giving hints

When asked now to prove the given implication, the tool will answer `No proof for epsilon`. Two things are possible: either the bound asked is too tight or the tool needs some additional rewriting hints. The first possibility can be excluded by asking `-> epsilon in ?` instead of the bound: Gappa will still give the same answer.

Thus additional hints must be given. How can these be found? A simple way is to add error term definitions for the different intermediate values and to ask Gappa their bounds. The tool will prove everything it is able to; for all the other bounds, hints can be found step by step. So we add:

```
epsilon0 = (zh - Z) / Z;
epsilon1 = (ZhSquarehl - MZSquare) / MZSquare;
epsilon2 = (polyUpper - MPolyUpper) / MPolyUpper;
epsilon3 = ((ZhSquareHalfhl + MZhzl) - MZSquareHalf) / MZSquareHalf;
```

```
[...]
epsilon11 = (Phl - MP) / MP;
```

and we ask now:

```
-> epsilon in ?
/\ epsilon0 in ?
/\ epsilon1 in ?
/\ epsilon2 in ?
/\ epsilon3 in ?
[...]
/\ epsilon11 in ?
```

The tool will answer:

```
No proof for epsilon
epsilon0 in [-1.11022e-16, 1.11022e-16]
epsilon1 in [-2.22045e-16, 2.22045e-16]
epsilon2 in [-7.50136e-16, 7.50136e-16]
epsilon3 in [-3.9402e+115, 3.9402e+115]
No proof for epsilon4
[...]
```

One continues by analyzing the results. The tool already gives a satisfying answer for ϵ_0, ϵ_1, and ϵ_2. Some other values are bounded by the tool, but the result is useless (ϵ_3 for example). And finally there are some expressions like $\epsilon 4$ that the tool is unable to bound.

Let us now find a hint for ϵ_3. By developing the definition of ϵ_3 and applying techniques 3. and 4. given in Section 4, we get

$$\begin{aligned}
\epsilon_3 &= \frac{(ZhSquareHalfhl + MZhzl) - MZSquareHalf}{MZSquareHalf} \\
&= \frac{\left(-\frac{1}{2}\cdot z_h^2 - z_h\cdot z_l\right) + \frac{1}{2}\cdot Z^2}{-\frac{1}{2}\cdot Z^2} \\
&= \frac{\left(-\frac{1}{2}\cdot z_h^2 - z_h\cdot z_l\right) + \frac{1}{2}\cdot (z_h+z_l)^2}{-\frac{1}{2}\cdot Z^2} \\
&= -\frac{z_l^2}{Z^2} \\
&= -\left(\frac{Z-z_h}{Z}\right)^2 \\
&= -\left(\frac{Z-Z\cdot(1+\epsilon_0)}{Z}\right)^2 \\
&= -\left(\frac{1-(1+\epsilon_0)}{1}\right)^2 \\
&= -\epsilon_0^2
\end{aligned}$$

Since we know now that $\epsilon_3 = -\epsilon_0^2$, we can give the corresponding hint to Gappa:

```
epsilon3 -> - epsilon0 * epsilon0;
```

and it will return a much better bound on ϵ_3:

```
epsilon3 in [-1.2326e-32, 1.2326e-32]
```

5.6 Completing the proof

By interactively iterating on the cycle of understanding what the tool is already capable of proving, and by developing hints, a complete proof can easily be established. Merely four more hints are needed for the given code. The order in which they are given to the tool mainly follows the control flow in the code to prove. Writing these hints is the hardest part of the proof work. They express all the optimizations and approximations the programmer has done when designing this numerical code.

One of these last hints, which we have not presented so far, is the `$` hint, which is used as follows:

```
Expression $ x;
```

Its meaning can be expressed as follows: "To get a good interval for `Expression`, try a dichotomy on `x`". This is also a general technique used in interval arithmetic.

The Gappa proof obtained is very concise: for our the 10-line C Code sequence that consists of 13 native double operations and 4 higher precision procedures, a Gappa file of about 100 lines is needed. Writing the Gappa file for the whole logarithm function was a matter of a few hours.

The tool computes the bounds in about 120 seconds on a recent machine (most of this time being spent in the dichotomy) and generates a formal proof for Coq of more than 4500 lines. If the proof had to be written in Coq by hand, it would probably require weeks of tedious work. Besides, most of this work would be lost if a part of the algorithm had to be rewritten. A Gappa description does not suffer from such a shortcoming: it can easily be adapted to a new implementation of the algorithm.

6 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 part of a C program 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 built-in rewriting rules, and an engine which explores the possible rewriting of expressions to minimize the size of the intervals. If decorrelation remains, Gappa allows one to provide new rewriting rules, but checks them. All this is well founded on a library of theorems which allow the obtained computation to be translated to a proof checkable by a lower-level proof assistant such as Coq and PVS. Finally, the tool can be questioned during the process of building the proof so that this process may be conducted interactively.

Therefore, it is possible to get quickly a fully validated proof with good confidence that this proof indeed proves property of the initial code. Gappa is by no means automatic: to apply it on a given piece of code requires exactly the same knowledge and cleverness a paper proof would. However, it requires much less work.

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 (0.4.11) is very stable and we may safely consider generalizing the use of this tool in the future developments of `crlibm`. It also took 6 months to develop a methodology and style well suited to the validation of elementary functions. This paper presented this aspect as well. Very probably, new problems will arise as we try to apply this methodology to new functions, so that it will need to be refined further.

References

[1] CR-Libm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/www/crlibm/.

[2] Standard 754-1985 for binary floating-point arithmetic (also IEC 60559), 1985. ANSI/IEEE.

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

[4] H. Brönnimann, G. Melquiond, and S. Pion. The Boost interval arithmetic library. In *Proceedings of the 5th Conference on Real Numbers and Computers*, pages 65–80, 2003.

[5] M. Daumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *6th Conference on Real Numbers and Computers*, pages 55–70, 2004.

[6] M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, Massachusetts, USA, 2005.

[7] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. Technical Report RR2005-37, LIP, September 2005.

[8] D. Defour. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, September 2003.

[9] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[10] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[12] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[13] J. Harrison. *The HOL Light manual*, 2000. Version 1.1.

[14] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq proof assistant: a tutorial: version 8.0*, 2004.

[15] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1973.

[16] Ch. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, September 2005.

[17] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[18] G. Melquiond and S. Pion. Formal certification of arithmetic filters for geometric predicates. In *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, 2005.

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

[20] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

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

[22] D.M. Priest. Fast table-driven algorithms for interval elementary functions. In *13th IEEE Symposium on Computer Arithmetic*, pages 168–174. IEEE, 1997.

[23] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE.

[24] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399