Verifying Code That Uses Error-Free Transformations

Tom Hubrecht
ENS de Lyon, CNRS, Inria, Université Lyon 1
LIP, UMR 5668
Lyon, France
tom.hubrecht@ens-lyon.fr

Guillaume Melquiond
Inria, CNRS, ENS de Lyon, Université Lyon 1
LIP, UMR 5668
Lyon, France
guillaume.melquiond@inria.fr

***Abstract*—The Gappa tool is designed to help developers verify some properties of their fixed- or floating-point algorithms, especially when it comes to rounding errors. Unfortunately, the tool is of little use when these algorithms extend the precision using some error-free transformations. This paper presents a small extension of the tool as well as a methodology to tackle the verification of some of these algorithms. For example, the tool is now able to generate a formal proof of a tight error bound for the double-word multiplication.**

***Index Terms*—Floating-point arithmetic, Error-free transformations, Formal verification, Automated reasoning**

I. Introduction

Floating-point arithmetic is a standard way of quickly performing numerical computations, at the cost of a limited precision. Therefore, evaluating the accuracy of these computations is usually a requirement, when one is concerned about the safety of some numerical code. In the context of critical systems where safety is paramount, a simple validation of the accuracy (*e.g.*, through testing) might not be sufficient; one might have to turn to mathematical proofs. While feasible, pen-and-paper proofs are usually tedious, so various tools have been designed to help developers with this endeavor [1, §13]. The Gappa prover is one of these tools; it is designed for the verification of short yet intricate fixed- and floating-point algorithms [2].

One way to improve the accuracy of an algorithm is to artificially extend the precision through multi-word arithmetic [1, §14]. State-of-the-art mathematical libraries like CORE-MATH make heavy use of multi-word arithmetic and have strong requirements on the correctness of their algorithms [3]. Unfortunately, all the aforementioned tools tend to assume that any floating-point expression that appears in the code is meaningful in isolation. As a consequence, they fall short when they have to analyze floating-point code that performs multi-word arithmetic, and thus leave developers empty-handed.

To bring some automation to the verification of multi-word floating-point algorithms, we have extended the Gappa prover with a set of specialized theorems that can be used to keep track of the overlap between the parts of a multi-word number as well as to quantify the relative contribution of the various rounding errors. Since Gappa is a tool with a rather steep learning curve, we have also devised a set of guidelines that allow for a smoother experience when verifying this kind of code. The motivation for building this work on top of Gappa is its flexible theorem-based architecture and its ability to generate formal proofs for the Rocq proof assistant, which is especially important in a domain where proofs are so error-prone.

The upsides and downsides of the presented approach will be evaluated on various examples of multi-word code, as they could appear in CORE-MATH. But this approach is of interest beyond this kind of use cases, as the added theorems are sufficiently generic to also improve the capabilities of Gappa in other scenarios.

The paper is organized as follows. Section II first reminds a few important facts about error-free transformations and multi-word arithmetic; it also explains how Gappa searches for proofs. Section III then presents the theorems we have added to Gappa. Section IV evaluates this extension over several examples of multi-word operators; it also describes the methodology we have devised to tackle these examples in a systematic way. Finally, Sections V and VI present some related works and conclude this paper.

II. Preliminaries

A. Error-free transformations

What makes it possible to compute with a larger precision than the working format is the following properties of floating-point arithmetic [1, §4]. First, when rounding to nearest in a given format, the rounding error of a floating-point addition is exactly representable in that format, assuming there is no overflow. Similarly, assuming there is neither underflow nor overflow, the rounding error of a floating-point multiplication is exactly representable in that format. Second, these errors are easily computed by a short sequence of floating-point operations. For instance, if an FMA operator is available, then the opposite e of the rounding error $\circ(x \cdot y) - x \cdot y$ can be computed with the following piece of C code:

```
double p = x * y, e = fma(x, y, -p);
```

For the rounding error of the floating-point addition, the code is slightly more complicated, but there are some optimized variants. Of particular interest is the FastTwoSum

algorithm, shown below. Assuming $|x| \geq |y|$, it computes $s = \circ(x + y)$ and $e = x + y - s$, when rounding to nearest.

```
double s = x + y, t = s - x, e = y - t;
```

To take advantage of these properties, we represent large precision numbers as the unevaluated sum of several floating-point numbers that do not overlap [1, §14]. For instance, a number x could be represented as $x_h + x_\ell$ with $|x_h| \gg |x_\ell|$, which is called a double-word number. To compute a double-word approximation of x^2, one could then compute the product x_h^2 as $u_h + u_\ell$, and then add to u_ℓ twice the floating-point product of x_h by x_ℓ. The corresponding C code is shown in Section IV-B.

B. Gappa

The Gappa tool is built upon a database of theorems that mostly deal with enclosures of real-valued expressions that contain rounding operators, *e.g.*, $\circ(x) + y \in [1, 2]$. The database contains roughly 300 theorems and their variants. Let us illustrate it using the following theorem, which is about addition and absolute value; it encompasses some knowledge about the triangular inequality.

$$\forall x, y \in \mathbb{R},\ \forall X, Y, Z \in \mathbb{I},\\ |x| \in X \wedge |y| \in Y \wedge P(X, Y, Z) \Rightarrow |x + y| \in Z.$$

Variables x and y denote real numbers, which, as far as Gappa is concerned are just expressions denoted by their abstract syntax trees (and so are $|x|$, $|y|$, and $|x+y|$). Variables X, Y, and Z denote real intervals, which are represented as pairs of floating-point numbers. The enclosures $|x| \in X$, $|y| \in Y$, and $|x+y| \in Z$ are the facts that Gappa tries to prove. Finally, the adhoc relation $P(X, Y, Z)$ ties all these enclosures together to make sure that the theorem is correct. This relation is computational in nature, *e.g.*, using interval arithmetic over X, Y, and Z. There are two ways to compute it. During the proof search, given X and Y, Gappa can compute an interval Z such that $P(X, Y, Z)$ holds. During the proof verification, given X, Y, and Z, the Rocq proof assistant is told how to formally check by computation that $P(X, Y, Z)$ holds, and hence that the theorem is correctly applied.

The proof search consists in repeatedly instantiating theorems from the database until the property given by the user has been proved. Mindlessly applying theorems would be inefficient, so the proof search is split into two stages. During the first stage, only the variables corresponding to real numbers are instantiated (x, y, and z above), as if backward reasoning was performed. In other words, Gappa starts from what it needs to prove and it deduces what it should prove first to do so. For instance, if Gappa needs to find an enclosure of the expression $|u/v + (t + w)|$, then it will instantiate the theorem using $x = u/v$ and $y = t + w$. Gappa keeps instantiating theorems until it cannot find any new instance.

During the second stage, Gappa considers all the instantiated theorems as inference rules and uses them to saturate the set of facts provided by the user. This time, the proof proceeds as if forward reasoning was performed. For instance, if at some point the set of facts contains $|u/v| \in [3, 4]$ and $|t+w| \in [1, 2]$, then Gappa can apply the partly instantiated theorem above with $X = [3, 4]$ and $Y = [1, 2]$, from which it computes Z and infers the new fact $|u/v + (t + w)| \in [1, 6]$. The interval computations are performed using the GNU MPFR library [4] with a default precision of 60 bits.

Since Gappa tries to refute the user formula, it stops once it finds an empty enclosure. If the user formula contains holes, it stops once it knows how to fill them to produce an empty enclosure. Gappa records a trace of all the theorems that were instantiated during the second stage. It uses this trace to optionally produce a formal proof that can be formally verified using the Rocq proof assistant, thanks to a library built on top of Flocq's comprehensive formalization of floating-point arithmetic [2]. This generated proof simply follows the steps of forward reasoning that were performed during the second stage of the proof search.

To make the first stage of the proof search more efficient, Gappa separates enclosures into several categories. The first one, which we will denote $\mathrm{BND}(x) \in J$, is the basic enclosure $x \in J$. The variant $\mathrm{ABS}(x) \in J$ corresponds to an enclosure $|x| \in J$ with the additional information that the lower bound of J is non-negative. To represent a relative error between $\tilde{x}$ and x, Gappa uses $\mathrm{REL}(\tilde{x}, x) \in J$. While this can intuitively be understood as the enclosure $\frac{\tilde{x}-x}{x} \in J$, this would cause issues in case of division by zero. Instead, Gappa defines it as follows:

$$\mathrm{REL}(\tilde{x}, x) \in J \Leftrightarrow \exists \varepsilon \in J,\ \tilde{x} = x \cdot (1 + \varepsilon).$$

Gappa supports more than just enclosures as facts. The most important ones for our purpose are $\mathrm{FIX}(x, k)$ which indicates that x is an integer multiple of 2^k, and $\mathrm{FLT}(x, k)$ which indicates that x fits on k bits. For example, if x is a binary64 number, both $\mathrm{FIX}(x, -1074)$ and $\mathrm{FLT}(x, 53)$ hold.

III. The LIN Predicate

The main idea of our approach is to extend Gappa's database with theorems about quotients. For that purpose, we introduce a new category of enclosures, named LIN, as it expresses linear dependencies between expressions. As with REL, we define it in a way that avoids divisions by zero, but for all intents and purposes, $\mathrm{LIN}(a, b) \in J$ can be understood as $\frac{a}{b} \in J$.

$$\mathrm{LIN}(a, b) \in J \Leftrightarrow \exists \lambda \in J,\ a = \lambda b\,.$$

An application of LIN is to mark some expressions as negligible relatively to some other ones, which is the case for the lower parts of a floating-point expansion compared to its most significant part, as is done with triple-word expansions in Section IV-C.

A. New theorems

To make effective use of this definition, we have added some simple theorems, of which a few examples are shown below. They are proved in a straightforward way by unfolding the definitions and applying basic properties of interval arithmetic.

Theorem 1 (Addition): If $\mathrm{LIN}(a,c) \in I$ and $\mathrm{LIN}(b,c) \in J$, then $\mathrm{LIN}(a+b,c) \in I+J$.

Theorem 2 (Product): If $\mathrm{LIN}(a,c) \in I$ and $\mathrm{LIN}(b,d) \in J$, then $\mathrm{LIN}(a \cdot b, c \cdot d) \in I \times J$.

Theorem 3 (Generalized relative error): If $\mathrm{LIN}(a,c) \in I$ and $\mathrm{REL}(b,a) \in J$, then $\mathrm{LIN}(b-a,c) \in I \times J$.

To illustrate the use of these theorems, let us consider the product of a double-word number $x_h + x_\ell$ by a floating-point number y, as implemented by the following C code:

```
double zh = xh * y, zl = fma(xl, y, fma(xh, y, -zh));
```

To bound the relative error between z_h+z_ℓ and $(x_h+x_\ell)\cdot y$, we need to compute an enclosure $\mathrm{LIN}(z_\ell - t, x_h \cdot y)$ with $t = x_\ell \cdot y - (z_h - x_h \cdot y)$. This can be obtained by applying Theorem 3 with $a = t$, $b = z_\ell$, and $c = x_h \cdot y$. In turn, this requires some enclosures $\mathrm{REL}(z_\ell, t)$ and $\mathrm{LIN}(t, x_h \cdot y)$. Assuming that the inner FMA is exact, the first enclosure is just the usual bound on the relative error of a rounding operator. The second enclosure can be obtained by applying Theorems 1 and 2, assuming we know an enclosure of $\mathrm{LIN}(x_\ell, x_h)$, which comes from the definition of a double-word number. Section IV shows larger examples of multi-word arithmetic.

B. Generalized theorems

A large part of Gappa's database is expressed as rewriting rules such as $((-a)-(-b)) = (-(a-b))$, for which Gappa generates the following theorem:

$$\mathrm{BND}(-(a-b)) \in J \Rightarrow \mathrm{BND}((-a)-(-b)) \in J.$$

This generation mechanism has been extended to provide equivalent theorems using the LIN predicate. From the previous rewriting rule, Gappa now also generates the following theorem:

Theorem 4: If $\mathrm{LIN}(-(a-b),c) \in J$, then $\mathrm{LIN}((-a)-(-b),c) \in J$.

Gappa's database also contained rewriting rules to simplify some specific quotients, *e.g.*,

$$\frac{(a+b)-(a+c)}{a+c} = \frac{b-c}{a+c}.$$

These have been dropped as they are now superseded by simpler, already existing rules, *e.g.*, $(a+b)-(a+c) = b-c$, for which Gappa generates the following, more generic theorem:

Theorem 5: If $\mathrm{LIN}(b-c,d) \in J$, then $\mathrm{LIN}((a+b)-(a+c),d) \in J$.

There were also some theorems with some contrived statements. For example, the original implementation of Sterbenz' lemma was stated using a REL enclosure:

$$\mathrm{REL}(a,b) \in [-\tfrac{1}{2},1] \wedge \mathrm{FLT}(a,p) \wedge \mathrm{FLT}(b,p') \Rightarrow \mathrm{FLT}(a-b,\max(p,p')).$$

The issue is that REL is meant to enclose relative errors, but neither a nor b can be seen as approximating the other one. The introduction of the LIN predicate allows us to express Sterbenz' lemma in the following, more classical way.

Theorem 6 (Sterbenz): If $\mathrm{LIN}(a,b) \in [\frac{1}{2},2]$ and $\mathrm{FLT}(a,p)$ and $\mathrm{FLT}(b,p')$, then $\mathrm{FLT}(a-b,\max(p,p'))$.

C. Usage of the magnitude

When dealing with a rounded number $\circ(x)$ and its associated error, two quantities are often used, either $\mathrm{ulp}(x)$ or $u \cdot |x|$ where $u := 2^{-p}$ with p the precision of the floating-point format. When rounding to nearest, assuming there is no underflow, the following two inequalities hold:

- $|\circ(x) - x| \leq u \cdot |x|$,
- $|\circ(x) - x| \leq \frac{1}{2}\mathrm{ulp}(x)$.

While the second bound is tighter, both inequalities have their use, and Gappa provides variants of them for the predicates BND, ABS, and REL. Let us explain how we ported these inequalities to LIN. But first, note that the second bound abruptly grows at powers of two, which might cause error bounds to be overestimated, as illustrated by the example of Section IV-A. To address that phenomenon, we have added the following theorem.

Theorem 7 (Rounding error, no underflow): Assume rounding to nearest and an unbounded exponent range. Assume $\mathrm{LIN}(a,b) \in [i_\ell, i_h]$ and $\mathrm{FLT}(b,1)$. Let $e \in \mathbb{Z}$ and $f \in [0,1)$ such that $\max(|i_h|,|i_\ell|) = 2^e \cdot (1+f)$. Then $\mathrm{LIN}(\circ(a)-a,b) \in [-2^{e'}, 2^{e'}]$ where e' is $e-p-1$ if $f \leq 2^{e-p-1}$, and $e-p$ otherwise.

The hypothesis $\mathrm{FLT}(b,1)$ is critical. It means that b has to be a power of two, which ensures, when the exponent range is unbounded, that $\circ(\frac{a}{b}) = \frac{\circ(a)}{b}$. There is also a simpler theorem in case we just need to bound $\circ(a)$ relatively to b.

Theorem 8 (Rounding): Assume rounding to nearest and an unbounded exponent range. If $\mathrm{LIN}(a,b) \in [i_\ell, i_h]$ and $\mathrm{FLT}(b,1)$ then $\mathrm{LIN}(\circ(a),b) \in [\circ(i_\ell), \circ(i_h)]$.

To deal with the case of a bounded exponent range and hence of potential underflow, we have added the following theorem, which is critical for the example of Section IV-F.

Theorem 9 (Rounding error): Assume rounding to nearest and a smallest positive floating-point number $2^{e_{\min}}$. If $\mathrm{LIN}(a,b) \in I$ and $\mathrm{ABS}(b) \in K$, then $\mathrm{LIN}(\circ(a)-a,b) \in J$ with $J = ([-2^{-p}, 2^{-p}] \times I) \cup ([-2^{-e_{\min}-1}, 2^{-e_{\min}-1}] \div K)$.

The various theorems that relate $\mathrm{LIN}(a,b)$ and $\mathrm{LIN}(\circ(a)-a,b)$ were especially tedious to prove and their formalization in Rocq was instrumental in finding the correct algorithms to compute the enclosures.

IV. Experiments

To illustrate the benefits of the LIN predicate (but also to see where it comes short), let us consider a few examples. Most of them are multi-word arithmetic operators, except for the last one which is of a completely different kind (Section IV-F).

All the examples will use a binary format with a precision of $p = 53$ bits (*e.g.*, the standard binary64 format) and rounding to nearest, ties breaking to even. The precision itself does not matter and Gappa would behave in the same way if another precision was chosen (except maybe for some very small precision, *e.g.*, $p \leq 3$). Therefore, error bounds will be expressed as a multiple of $u = 2^{-53}$. To improve readability,

higher-order terms, if present, will be summarized as just "ε". For instance, $7u^2+\varepsilon$ means a bound of the form $7u^2+ku^3$ with k negligible with respect to u^{-1}, but Gappa does provide k.

In all the examples, the main rounding operator will be denoted `rnd` in the Gappa scripts. If the format is the same as binary64 (except that numbers can grow arbitrarily large without overflowing), the rounding operator will be defined at the start of the Gappa script as follows:

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

If the numbers can become arbitrarily close to zero (or equivalently, if no underflow did occur in binary64), the rounding operator will be defined as follows:

```
@rnd = float<53,ne>;
```

For most of the examples below, Gappa produces a proof instantly. It is only for the half double-word addition (Section IV-D) that the tool needs to work a few seconds. For all the examples, the produced formal proof is rather short and successfully checked by the Rocq proof assistant.

A. Double-word multiplication

The first example is the algorithm for multiplying two double-words $x_h + x_\ell$ and $y_h + y_\ell$, assuming that the FMA operator is available. It could be implemented using the following C code:

```
double uh = xh * yh, ul = fma(xh, yh, -uh);
double v0 = fma(xh, yl, ul), v1 = fma(xl, yh, v0);
double zh = uh + v1, zl = v1 - (zh - uh);
```

The goal is to obtain a bound on the relative error between $z_h + z_\ell$ and $(x_h + x_\ell) \times (y_h + y_\ell)$, assuming the inputs are non-overlapping expansions, that is, $x_h = \circ(x_h + x_\ell)$ and $y_h = \circ(y_h + y_\ell)$. As is usual in the literature for this kind of algorithm, we will assume that no underflow can occur and define `rnd` accordingly.

This assumption also means that the FMA used to compute u_ℓ is actually exact. When modeling the algorithm in Gappa, we will therefore define u_ℓ as just $x_h y_h - u_h$. Avoiding rounding operators whenever possible reduces the proof effort for Gappa, but more importantly, it also reduces our modeling work. Indeed, we will have to provide hints to guide Gappa, and the less rounding operators there are, the easier the hints will be to state.

Actually, $x_h y_h - u_h$ is not the best way to define u_ℓ. Indeed, Gappa's heuristics during the first stage of the proof search rely on the syntax of terms and most of them expect a rounding error to be expressed as $\circ(a) - a$. Therefore, the best definition for u_ℓ is $-(u_h - x_h y_h)$ since $u_h = \circ(x_h y_h)$.

Another instance for this kind of transformation is the definition of z_ℓ, which is computed using a FastTwoSum sequence. Again, rather than using rounding operators, it is better to express z_ℓ as just $-(z_h - (u_h + v_1))$. This assumes that the prerequisite $|u_h| \geq |v_1|$ of the FastTwoSum routine is satisfied, but that is something we could ask Gappa to prove, if needed.

For the example of the double-word multiplication, we do not even need to define z_h and z_ℓ. Indeed, assuming that we have proved $|u_h| \geq |v_1|$, we have $z_h + z_\ell = u_h + v_1$. So, the relative error we are interested in is the one between $u_h + v_1$ and $(x_h+x_\ell)\times(y_h+y_\ell)$. The last modeling step is to introduce some real numbers x and y, and to define x_h as $\circ(x)$, x_ℓ as $-(x_h - x)$, and similarly for y_h and y_ℓ. Therefore, the Gappa script starts as follows:

```
@rnd = float<53,ne>;
xh = rnd(x); xl = -(xh - x);
yh = rnd(y); yl = -(yh - y);
uh = rnd(xh * yh); ul = -(uh - xh * yh);
v0 = rnd(ul + xh * yl);
v1 = rnd(v0 + xl * yh);
```

If we were to ask Gappa for the relative error, we would not get any interesting result. So, we need to guide the tool toward the correct proof. In particular, we need to explain to Gappa that, if not for the rounding errors and the ignored term $x_\ell y_\ell$, the computed value $u_h + v_1$ would actually be equal to $x + y$. This is best expressed by adding the following line at the end of the script:

```
uh + (ul + xh*yl + xl*yh) - x*y -> - xl*yl;
```

The hint above is akin to an axiom, so any mistake would have disastrous consequences. Fortunately, Gappa double-checks that, given the definitions of u_ℓ, x_ℓ, and y_ℓ, both sides of the arrow are actually equal.

We have now provided enough information to Gappa and we can ask for an error bound. Let us start by constraining x and y to some small interval $[1, 2]$ to see how it behaves:

```
{ x in [1,2] /\ y in [1,2] -> uh+v1 -/ x*y in ? }
```

The notation "`a -/ b in ?`" just asks Gappa for the enclosure $\mathrm{REL}(a, b)$.

Without support for LIN, Gappa answers that the relative error is bounded by $7u^2+\varepsilon$, which is rather good but not quite the best known error bound. So, let us ask Gappa to split the input intervals into smaller pieces by adding the following hint at the end of the script:

```
$ x, y;
```

We now get the bound $5u^2$ (and not $5u^2 + \varepsilon$), which is the best known bound, but it is formally proved by Gappa only for $x, y \in [1, 2]$. Let us now remove the constraints on x and y. We will just keep the fact that they are nonzero. Not only will we ask Gappa to bound the relative error, but we will also ask Gappa how small v_1 is with respect to u_h (to check whether the FastTwoSum precondition holds).

```
{ x <> 0 /\ y <> 0 ->
  uh+v1 -/ x*y in ? /\ v1 // uh in ? }
```

The notation "`a // b in ?`" just asks Gappa for the enclosure $\mathrm{LIN}(a, b)$.

The hint about dividing the input intervals into smaller ones is no longer relevant, as there are no input intervals anymore. More generally, the example is complicated enough so that, without LIN, Gappa cannot say anything about the relative error. But thanks to the work presented in this paper, Gappa now succeeds and proves the bound $6u^2 + \varepsilon$. Here, it is mostly a matter of applying Theorem 2 to get an enclosure

of LIN$(x_\ell y_\ell, xy)$. Not only is the bound quite good, but it is now proved valid for the whole domain rather than a single binade. Gappa also proves that $|v_1/u_h|$ is smaller than $3u+\varepsilon$, *i.e.*, the overlap between v_1 and u_h is reduced to a few bits.

While this is encouraging, the error bound $6u^2+\varepsilon$ is still worse than the expected bound $5u^2$. To recover it, we need to instruct Gappa to split the input intervals into smaller intervals, as we did in the case of the binade $[1,2]$. Therefore, we look for some simple function f such $x/f(x)$ is always in the interval $[1,2]$. Fortunately, we can use the rounding toward infinity with a precision of just one bit. As it is a rounding operator, the existing machinery of Gappa supports it properly and we do not need to provide any new theorem. We define this operator at the start of the script and we name it `ufp`, as it plays a role reminiscent of the *unit-in-the-first-place*:

```
@ufp = float<1,aw>;
```

Since the result of ufp fits on one bit, Gappa will be able to use the improved theorems of Section III-C. We just need to tell the tool how to relate the various expressions at hand.[1]

```
(uh + v1 - x * y) / (x * y) ->
  ((uh + v1 - x * y) / (ufp(x) * ufp(y)))
    / ((x / ufp(x)) * (y / ufp(y)));
```

All that is left is to instruct Gappa to consider smaller intervals. We express it as before, except that the hint is now about $x/\text{ufp}(x)$ and $y/\text{ufp}(y)$ rather than just x and y.

```
$ (x / ufp(x)), (y / ufp(y));
```

Instead of letting Gappa blindly split the intervals, we could be more directive and use the following hint instead. This tells Gappa to split the enclosure of $(x/\text{ufp}(x)) \cdot (y/\text{ufp}(y))$ into sub-intervals on which $|\text{REL}(u_h+v_1, xy)| \leq 5\cdot 2^{-106}$ holds. This halves the size of the generated proof.

```
|uh + v1 -/ x * y| <= 5b-106 $
   (x / ufp(x)) * (y / ufp(y));
```

Whichever hint is used, Gappa successfully proves that the relative error is less than $5u^2$. Figure 1 shows the whole script. Note that we never expressed that x_ℓ and y_ℓ are floating-point numbers; the property holds even if x and y are not actual double-word numbers.

B. Double-word squaring

The double-word multiplication can easily be optimized for the squaring case. The goal is now to bound the relative error between u_h+v and $(x_h+x_\ell)^2$ for the following code:

```
double uh = xh * xh, ul = fma(xh, xh, -uh);
double v = fma(2 * xh, xl, ul);
double zh = uh + v, zl = v - (zh - uh);
```

This second example is handled in the same way as the double-word multiplication. Since the script is similar to the one of Figure 1, it will not be shown here. What makes this example interesting is that the relative error is no longer

[1]Contrary to the hint that indicated how u_h+v_1 and xy were related, this one does not carry much information and should presumably be inferred automatically by Gappa.

```
@ufp = float<1,aw>;
@rnd = float<53,ne>;

xh = rnd(x); xl = -(xh - x);
yh = rnd(y); yl = -(yh - y);
uh = rnd(xh * yh); ul = -(uh - xh * yh);
v0 = rnd(ul + xh * yl);
v1 = rnd(v0 + xl * yh);

{ x <> 0 /\ y <> 0 ->
  uh+v1 -/ x*y in ? /\ v1 // uh in ? }

uh + (ul + xh*yl + xl*yh) - x*y -> - xl*yl;

(uh + v1 - x * y) / (x * y) ->
  (uh + v1 - x * y) / (ufp(x) * ufp(y))
    / ((x / ufp(x)) * (y / ufp(y)))
{ x <> 0, y <> 0, ufp(x) <> 0, ufp(y) <> 0 };

|uh + v1 -/ x * y| <= 5b-106 $
  (x / ufp(x)) * (y / ufp(y));
```

Fig. 1. Full Gappa script for the double-word multiplication.

symmetric. Indeed, the neglected term is now x_ℓ^2 (instead of $x_\ell y_\ell$), which is of constant sign, thus skewing the error. As a consequence, when the input x is constrained to $[1,2]$, Gappa is able to prove that the relative error is contained in the interval $[-3u^2; 2u^2+\varepsilon]$.

Thanks to the support for LIN, the constraint on x can be removed, in which case Gappa proves that the relative error is smaller than $4u^2+\varepsilon$. As before, we can introduce ufp(x) to give a way for Gappa to split the input interval. This reduces the bound to $3u^2$. In both cases, the asymmetry of the relative error has been lost, as Gappa no longer takes advantage of the fact that x_ℓ^2 is non-negative. This could be fixed by adding a dedicated theorem for proving LIN(a^2, b^2).

C. Triple-word multiplication

Let us now consider the case of a triple-word multiplication, *i.e.*, the product of $x_h+x_m+x_\ell$ by $y_h+y_m+y_\ell$. It could be implemented using the following C code [5, Algorithm 9]. Note that we omit the last step of the algorithm, which is to normalize the 5-word expansion $u_h+u_\ell+b+c+z_0$ and to drop the two least significant terms.

```
double uh = xh * yh, ul = fma(xh, yh, -uh);
double v0h = xh * ym, v0l = fma(xh, ym, -v0h);
double v1h = xm * yh, v1l = fma(xm, yh, -v1h);
double w0 = fma(xh, yl, v0l);
double w1 = fma(xl, yh, v1l);
double z0 = w0 + w1;
double b = v0h + v1h, b2 = add3(v0h, v1h, -b);
double c = fma(xm, ym, b2);
```

Our goal is to bound the relative error between $u_h+u_\ell+b+c+z_0$ and $x\cdot y$. There are a few important differences with respect to the example of the double-word multiplication shown in Figure 1. The first one is that the three pieces of a triple-word expansion are not as neatly separated as in the double-word case. There might be some overlap between some of them, depending on the way the expansion was normalized. Therefore, we will make use of the LIN predicate to state

```
@ufp = float<1,aw>;
@rnd = float<53,ne>;

x = xh + xm + xl;
y = yh + ym + yl;

uhl = xh * yh;
v0h = rnd(xh * ym); v0l = -(v0h - xh * ym);
v1h = rnd(xm * yh); v1l = -(v1h - xm * yh);
b = rnd(v0h + v1h); b2 = -(b - (v0h + v1h));
c = rnd(b2 + xm * ym);
w0 = rnd(v0l + xh * yl);
w1 = rnd(v1l + xl * yh);
z0 = rnd(w0 + w1);

{ |xm//ufp(xh)|<=1b-53 /\ |xl//ufp(xh)|<=1b-106 /\
  |ym//ufp(yh)|<=1b-53 /\ |yl//ufp(yh)|<=1b-106 /\
  xh <> 0 /\ yh <> 0 ->
  uhl + b + c + z0 -/ x * y in ? }

(uhl + b + c + z0) - x*y ->
  - xm*yl - xl*ym - xl*yl + (c - (b2 + xm*ym))
  + (z0 - ((v0l + xh*yl) + (v1l + xl*yh)));

((uhl + b + c + z0) - x * y) / (x * y) ->
  (((uhl+b+c+z0) - x * y) / (ufp(xh) * ufp(yh)))
    / ((x / (ufp(xh)) * (y / ufp(yh))))
{ ufp(xh) * ufp(yh) <> 0 };
```

Fig. 2. Full Gappa script for the triple-word multiplication.

how x_m and x_ℓ are located with respect to $\mathrm{ufp}(x_h)$, following Priest's definition [5].[2]

```
|xm//ufp(xh)| <= 1b-53 /\ |xl//ufp(xh)| <= 1b-106
```

The second interesting point is the hint we provide to indicate which products were neglected, as well as the proper way to handle z_1. As always, both sides of the arrow are equal, which Gappa double-checks.

```
(uhl + b + c + z0) - x*y ->
  - xm*yl - xl*ym - xl*yl + (c - (b2 + xm*ym))
  + (z0 - ((v0l + xh*yl) + (v1l + xl*yh)));
```

Finally, we also need to tell the tool how to go from an error relatively to $\mathrm{ufp}(x_h)\cdot\mathrm{ufp}(y_h)$ to the relative error we are looking for, in a way reminiscent of what was done for the double-word multiplication. The full Gappa script is shown in Figure 2. The tool successfully proves that the relative error is bounded by $28u^3+\varepsilon$. Bisection does not improve it, so it is not used. Note that this bound does not account for the terms that are discarded after the final normalization, so it should be increased accordingly (by about $2u^3+\varepsilon$).

Note that Gappa makes it easy to experiment with variants of the algorithm. For example, we can modify the last two lines as follows.

```
// double b, b2 = twosum(v0h, v1h, &b);
// double c = fma(xm, ym, b2);
double z1 = fma(xm, ym, z0);
```

This time, it is a matter of bounding the relative error between $u_h + u_\ell + v_{0h} + v_{1h} + z_1$ and $x \cdot y$, for which

[2]The bounds might seem too small to allow for overlap, but keep in mind that ufp is twice as large as its usual definition.

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

xh = rnd(x); xl = -(xh - x);
y = rnd(y_);
zh = rnd(xh + y); v = -(zh - (xh + y));
zl = rnd(xl + v);

{ x in [1,2] /\ |y| <= 1b1024 /\
  |x + y| >= 1b-1074 /\
  @FIX(x,-1074) /\ @FLT(xl,53) /\ @FLT(v,53) ->
  zh + zl -/ (x + y) in ? }

(zh + zl) - (x + y) -> zl - (xl + v);
xl + -((xh + y) - (xh + y)) -> xl;

|zh + zl -/ x + y| <= 0x1.0000000000001p-105 $
   |y| in (10), xh + y;
```

Fig. 3. Full Gappa script for the half double-word addition.

Gappa gives $32u^3+\varepsilon$. This makes it a bit less accurate than the original algorithm, but more accurate and possibly also faster than Fabiano, Muller, and Picot's "faster version" [5, Algorithm 10].

D. Half double-word addition

Let us now consider the following piece of code,[3] which computes the double-word sum of a double-word x_h+x_ℓ and a floating-point number y.

```
double zh = xh + y;
double v = add3(xh, y, -zh);
double zl = xl + v;
```

As before, the goal is to obtain a bound on the relative error. Here it is between $x_h + x_\ell + y$ and $z_h + z_\ell$. The main issue is the possibility of dramatic cancellation when y is close to $-x_h$. Without support for LIN, Gappa is able to prove an error bound of $2u^2+\varepsilon$ for $x \in [1,2]$ and $|y| \leq 10$ and $|x+y| \geq 2^{-1074}$. (The last hypothesis is just a way to forbid the trivial case $x_h = -y$ and $x_\ell = 0$, as it confuses Gappa.) But since the proof relies on a bisection, the larger the domain of y, the longer it takes.

The LIN predicate does not help when cancellation can occur and we still need to perform a bisection. But it allows Gappa to tackle the case $x \in [1,2]$ and $|y| \geq 10$, where it finds an error bound of $\frac{9}{8}u^2+\varepsilon$. By telling Gappa to consider the cases $|y| \leq 10$ and $|y| \geq 10$ separately ("`$ |y| in (10)`"), it succeeds in proving that the relative error is bounded by $2u^2+\varepsilon$ for $x \in [1,2]$. The full Gappa script is shown on Figure 3. Notice how we had to tell the tool that x is a multiple of 2^{-1074} ("`@FIX(x,-1074)`") and that x_ℓ and v fit on 53 bits ("`@FLT(..,53)`"), as there is not enough information to deduce these facts otherwise.

This is enough to prove the bound in the general case, *i.e.*, when x is unbounded, as modulo multiplying y by the sign of x and dividing it by the appropriate power of two, we fall into one of the cases proved by Gappa.

[3]The `add3` function performs the sum of three floating-point numbers with just one final rounding.

E. Double-word addition

Gappa, however, is unable to tackle more complex summation algorithms. As an example, consider the case of the `madd2` algorithm [6]. The bound proved by the authors is $2u^2$, which Gappa can only prove on easy inputs (*e.g.*, $x \in [1,2], y \in [0,2]$) with bisection. But as soon as cancellation can occur (*e.g.*, $x \in [1,2]$ and $y \in [-2,2]$), Gappa is lost. If we limit the amount of cancellation (*e.g.*, $|x+y| \geq 2^{-50}$), then Gappa can recover an error bound of $3u^2+\varepsilon$, again using a bisection.

As it stands, the support for the LIN predicate does not bring much. The only advantage is that, for slightly less tight error bounds, it allows Gappa to find proofs without having to perform a bisection. For example, when the amount of cancellation is limited by $|x+y| \geq 2^{-50}$, Gappa can prove a bound $\frac{17}{4}u^2 + \varepsilon$ (*i.e.*, twice as large as the optimal one). Without LIN, Gappa would need a bisection to find a useful result.

F. Homogeneous functions

The last example revisits one of the earliest floating-point algorithms verified using Gappa [7]. It computes the orientation (clockwise, counter-clockwise, or aligned) of three points (x_i, y_i) in the 2D plane, which is widely used in computational geometry, *e.g.*, to compute Delaunay triangulations. This orientation would be given by the sign of r in the code C below, if it was computed without any rounding error. While it is critical that the returned sign is always correct, the function is allowed to give up, in which case a more precise, much slower algorithm will be used. To detect whether the sign of r might have been incorrectly computed, the code also computes a dynamic bound e on the absolute error of r.

```
double x12 = x2 - x1, x13 = x3 - x1,
       y12 = y2 - y1, y13 = y3 - y1;
double mx = fmax(fabs(x12), fabs(x13)),
       my = fmax(fabs(y12), fabs(y13));
double e = fmax(mx * my, 0x1.0p-971)
         * 0x1.8000000000003p-51;
double r = (x2 - x1) * (y3 - y1)
         - (x3 - x1) * (y2 - y1);
if (r < -e) return NEGATIVE;
if (r > +e) return POSITIVE;
return UNKNOWN; // need a more precise algorithm
```

Proving the correctness of the code therefore amounts to proving that e is indeed such a bound, which amounts to proving that the constants that appear in its definition are large enough. But these constants should be as small as possible to make sure that the code does not give up too often. The full Gappa script,[4] for which no hints are needed, is shown in Figure 4. The tool automatically proves that $(r-R)/(m_x \cdot m_y)$ is smaller than $6u+\varepsilon$.

Note that the literals in the definition of e also need to account for the rounding errors that occur during the evaluation of e itself. Proving that these constants are large enough is a classical use of Gappa and hence not shown here.

[4]Thanks to gradual underflow, the last subtraction cannot spuriously change the sign of r, so its rounding error is ignored in the script.

```
@rnd = float<ieee_64,ne>;
x1 = rnd(x1_); x2 = rnd(x2_); x3 = rnd(x3_);
y1 = rnd(y1_); y2 = rnd(y2_); y3 = rnd(y3_);

x12 = rnd(x2 - x1); x13 = rnd(x3 - x1);
y12 = rnd(y2 - y1); y13 = rnd(y3 - y1);
r = rnd(x12 * y13) - rnd(x13 * y12);
R = (x2-x1) * (y3-y1) - (x3-x1) * (y2-y1);

{ x12 // mx in [-1,1] /\ x13 // mx in [-1,1] ->
  y12 // my in [-1,1] /\ y13 // my in [-1,1] ->
  mx * my in [1b-1022,1b2048] ->
  r - R // mx * my in ? }
```

Fig. 4. Full Gappa script for the 2D orientation predicate.

V. Related Works

There are few tools like Gappa that are able to analyze floating-point algorithms and produce formal proofs that can be mechanically checked. One of them is FPTaylor [8]. It works by abstracting an expression into a symbolic term representing its infinitely precise value, a linear combination with symbolic coefficients of the rounding errors that taint it, and a numerical interval to account for all the remaining non-linear error terms (*i.e.*, a symbolic zonotope). A normal use of FPTaylor would certainly fail on the kind of algorithms we tackle in this work. Indeed, despite the symbolic computations, there would be a point at which the numerical interval would need to be filled with unbounded values. Possibly, making this numerical interval relative to some simple symbolic expression (as is done with the LIN predicate in this paper) could avoid this issue. FPTaylor would also have to be told which variables are actually proportional to rounding errors, which is the key staple of error-free transformations.

An approach that has been gaining traction recently is the SMT-LIB theory of floating-point arithmetic [9], as it is now supported by widely used SMT solvers such as CVC5 and Z3. It is usually implemented using bit-blasting, that is, floating-point operators are represented by their boolean circuits and the whole problem is then handled by a SAT solver. In some way, using bit-blasting amounts to doing an exhaustive testing of all the inputs, but with large shortcuts when recurring test patterns are detected. The combinatorial explosion obviously becomes an issue as the number of operators or the precision increase, but there is a more important concern. A bit-blasting approach is only meaningful when the property to prove can easily be expressed as a boolean circuit. This is for example the case when one wants to prove that a rounded operation is actually exact. But error bounds are a whole different matter.

Despite these shortcomings of the SMT solvers, Zhang and Aiken have reached a significant milestone when it comes to the automatic verification of multi-word adders [6]. They have devised a dedicated abstraction (*i.e.*, a set of theorems) for floating-point adders and their rounding errors, verified its soundness using the floating-point theory of SMT solvers, and then used it to verify multi-word adders, again using SMT solvers to carry the proof. In some way, their layered approach

is similar to ours, except that our abstraction is verified using the Rocq proof assistant and that Gappa comes with its own proof engine. Therefore, the main difference between both approaches lies in the chosen abstraction. Their experiments show that their abstraction works very well to prove error bounds on multi-word adders (and also multi-word multipliers, as they can be represented as adders), while Gappa is still lacking, despite the presented work. But it is not clear whether their abstraction has any use outside the very specific domain of multi-word adders. Moreover, though the proofs produced by SMT solvers can be checked independently by other SMT solvers, gaining insights about the obtained bounds is not convenient, in contrast to Gappa proofs.

Before Zhang and Aiken's work, algorithms for normalizing expansions, which can be seen as a very specific kind of multi-word adders, have been formally verified by hand [10]. The lack of any dedicated automation makes it a tremendous proof effort ("unexpectedly complicated") but there are some advantages to it. In particular, the algorithms are proved once and for all, whatever the precision or the number of words. Such a parametricity is completely out of Gappa's scope and our work does not help in any way.

Finally, regarding homogeneous predicates, the original way of verifying their correctness was by extending Gappa with adhoc arithmetic operators and their associated theorems [7]. Therefore, the large gap between what was fed to Gappa and the actual algorithms required a careful analysis of the generated proofs to convince oneself that the algorithms were actually correct. Our approach shown in Section IV-F is much more satisfying. Indeed, as it relies on the standard rounding operators and formats, the produced proofs are actual correctness proofs of the algorithms and formally checking them with the Rocq proof assistant becomes meaningful.

VI. Conclusion

This paper has presented an extension of the Gappa prover to ease the proof of algorithms that rely on multi-word arithmetic. Implementing this extension did not require much changes to the tool, as its proof engine already supported the REL predicate, which is quite similar to LIN. So, it was mostly a matter of extending Gappa's database of theorems. Also, some existing theorems that were abusing REL were rephrased to use LIN. Presumably, the REL predicate could now be dropped entirely, as LIN is strictly more expressive, but this would be a much more invasive change to Gappa. Since we have formally verified all the new theorems using the Rocq proof assistant, our modified version of Gappa offers the same guarantees as the original one.

We have also proposed a methodology to ease the automatic verification of these algorithms. The two main tricks are the way to represent basic blocks such as the FastTwoSum algorithm as well as the way to guide the tool toward a reasoning based on the unit-in-the-first-place. Other than that, it is just a matter of pointing where the error comes from, as we would do on paper, except that Gappa will take care of all the tedious and error-prone details. As shown with the example of the triple-word multiplication in Section IV-C, the methodology is especially effective. One can easily experiment with new implementations and immediately obtain a formal proof of correctness. One can even experiment with different amount of overlaps, in case one needs some adhoc implementation.

While our original motivation for the LIN predicate was to deal with overlaps in multi-word arithmetic, it also helps in verifying floating-point algorithms that are completely unrelated, as shown in Section IV-F. In that case, Gappa does not even need any hint; the proof is fully automatic.

While our approach was extremely successful for verifying multi-word multipliers, it does not help that much for multi-word adders, especially when there are cancellations. It is not clear yet to us whether this is just a matter of adding some more theorems about the LIN predicate or whether a completely different approach is needed, as exemplified by the work of Zhang and Aiken [6]. That being said, in some specific contexts (*e.g.*, multi-word operators used during the evaluation of an elementary function), the inputs of the adders are sufficiently constrained for Gappa to still be of use.

Another venue to explore is related to the proposed methodology. Indeed, at no point does the user have anything clever to do. Thus, most of it could be handled by the tool itself: recognizing code patterns such as the FastTwoSum, considering the use of the ufp, and so on. More generally, it would be interesting to see whether the tool could be made to recognize floating-point expansions, that is, which expressions should be checked for overlaps and combined into sums.

References

[1] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser Basel, 2018.

[2] S. Boldo and G. Melquiond, "Some formal tools for computer arithmetic: Flocq and Gappa," in *28th IEEE International Symposium on Computer Arithmetic*, M. Joldes and F. Lamberti, Eds., Jun. 2021.

[3] A. Sibidanov, P. Zimmermann, and S. Glondu, "The CORE-MATH project," in *29th IEEE International Symposium on Computer Arithmetic*, S. Oberman, B. Pasca, and L. Sousa, Eds., Sep. 2022, pp. 26–34.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13:1–13:15, Jun. 2007.

[5] N. Fabiano, J.-M. Muller, and J. Picot, "Algorithms for triple-word arithmetic," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1573–1583, Nov. 2019.

[6] D. K. Zhang and A. Aiken, "Automatic verification of floating-point accumulation networks," in *37th International Conference on Computer Aided Verification*, Jul. 2025, pp. 215–237.

[7] 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.

[8] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, pp. 2:1–2:39, Dec. 2018.

[9] P. Rümmer and T. Wahl, "An SMT-LIB theory of binary floating-point arithmetic," in *International Workshop on Satisfiability Modulo Theories*, 2010. [Online]. Available: https://www.cprover.org/SMT-LIB-Float/smt-fpa.pdf

[10] S. Boldo, M. Joldes, J.-M. Muller, and V. Popescu, "Formal verification of a floating-point expansion renormalization algorithm," in *8th International Conference on Interactive Theorem Proving*, Sep. 2017, pp. 98–113.