

Generating certified properties for numerical expressions and their evaluations^{*}

Marc Daumas¹ and Guillaume Melquiond²

¹ LIRMM (UMR 5506 CNRS-UM2), visiting LP2A (EA 3679 UPVD)

² LIP (UMR 5668 CNRS-ENS Lyon-INRIA)

Abstract. We present Gappa, a tool that can generate certified properties based on dyadic fractions, interval arithmetic and forward error analysis. Gappa operates on numerical expressions and on their evaluation on computers. For each property, Gappa generates a proof that can be checked with an automatic proof checker with the help of a companion library of verified facts. So far, Gappa generates proofs for either Coq and HOL Light and we have developed a large companion library for Coq dealing with the addition, multiplication, division, and square root, in fixed- and floating-point arithmetics. Gappa handles seamlessly additional properties expressed as interval properties or rewriting rules in order to establish more intricate results. Users can simultaneously provide bounds to be proved on expressions and ask Gappa to propose ones on other expressions. Recent work has proved that Gappa is perfectly adapted to the verification of small pieces of software. For larger pieces of software, Gappa can either be used to double check assertions produced by non verified tools or be invoked as needed by tools that handle loops and branches but miss the ability to handle possible effects of the accumulation and magnification of negligible errors.

1 Introduction

Gappa is a simple and efficient tool for automatically developing certified properties and proofs in general calculus, in computer arithmetic [1], and in the engineering of numerical software [2,3] and hardware [4]. These properties deal with arithmetic expressions on real and rational numbers and their evaluation in computers on fixed- and floating-point data formats.

Software and hardware designers usually answer questions such as: **1.** How accurate are results? **2.** Will hardware or software exhibit any exceptional behavior? The answer to question **1** is very often “sufficiently” [5] and the answer to question **2** is very often “never” [6].

When we deal with programs that involve model, truncation and/or round-off errors, we cannot expect programs to yield exact results, but specifications should still fully characterize the accuracy of the results. Moreover specifications

^{*} This work has been partially founded by PICS 2533 of the CNRS and projet EVA-Flo of the ANR.

normally consider that the exceptional behavior of one operation (such as a division by zero, an overflow or an invalid operand) correctly handled by the rest of a program leading to meaningful results is not an exceptional behavior of the program.

Writing a complete and accurate specification about the behavior of some numerical software is usually a difficult task requiring some familiarity with backward error analysis, first order analysis, condition number and singular value decomposition [7,8]. Such work is even more repulsive to designers as it leads to a dead-end similar to what was mentioned satirically in [9]. To the author's best knowledge, Gappa is the first tool both able to automatically handle some of the properties encountered in the specification of numerical software and based on sufficiently strong foundations to become part of future tools routinely used in software engineering.

Gappa is well suited to certify numerical programs appearing in safety critical applications such as air transportation or ubiquitous software such as basic libraries approximating the common elementary functions (sin, exp, *etc*). After each invocation, Gappa generates a certificate that is a formal proof that can be checked independently. Similar methodology has proved to be sufficient to meet the highest Common Criteria Evaluated Assurance Level (EAL 7) [10,11] and it may now be applied to numerical applications using floating- and fixed-point arithmetics.

Properties that are most often needed involve: the range of variables appearing in programs to prevent any exceptional behavior (overflow or division by zero) and the range of absolute and/or relative errors to characterize the accuracy of results.

Two other projects are currently mixing interval arithmetic and automatic proof checking [12,13]. Both projects focus on providing tools to perform interval arithmetic within an automatic proof checker, ACL2 for the first one and PVS for the second one. Our goal is to provide *invisible formal methods* [14] in the sense that Gappa delivers formal certificates to users that are not expected to ever write any piece of proof in any formal proof system. We use Coq proof assistant [15,16], but ongoing work shows that Gappa can work with other proof assistants such as PVS and HOL Light.

The continuing work on interval arithmetic [17,18] has created a huge set of useful techniques to deliver accurate answers in a reasonable time. Each technique is adapted to a specific class of problems and most evaluations yield accurate estimations only if they are handled by the appropriate techniques in the appropriate order. Blending ranges and properties on dyadic fractions has also been heavily used in computer arithmetic and [19] is one recent noticeable reference.

Our goal in developing Gappa is to provide a tool that is able to consider many techniques using interval arithmetic, dyadic fractions, and rewriting rules. Gappa is able to follow hints when some are available (either given by an heuristic or by the user), and it otherwise performs an exhaustive search. Once it has produced a valid proof, Gappa simplifies it in order to reduce the certification

time, as in-depth proof checking is and will remain much slower than simple C++ evaluation.

We first describe how to write an efficient script for Gappa. We then present how Gappa works with proof checkers, extending [20]. We finish this report with perspectives, experiments and concluding remarks.

2 Input scripts to Gappa

Gappa is composed of an independent program written in C++, based on Boost interval arithmetic library [21] and MPFR [22], and a companion library of Coq theorems. Gappa produces a Coq file for a given input script including properties to prove. The file contains proofs of the properties. Validity of proofs can then automatically be checked by Coq.

The input file to Gappa contains a set of hypotheses each stating that a variable or an expression is within an interval. Gappa handles basic arithmetic operators (addition, subtraction, multiplication, division, and square root) and the support library contains theorems so these operators can be used in proofs.

Gappa input file also contains goals using the same format as hypotheses. Consider for example that y is the result of a program. We may define Y (upercase) as the exact answer without any model, truncation or rounding error. We will certainly be interested in

- an interval containing y to guarantee that the result does not overflow,
- an interval containing $y - Y$ or $(y - Y)/Y$ to guarantee that the result is accurate.

Intervals in goals may be replaced by question marks when Gappa should propose some enclosing intervals. Users cannot use question marks for intervals that appear as hypotheses in the logical formula.

Warning messages, error messages, and results are displayed on the standard error output. Gappa sends to the standard output a formal proof of the logical formula; its format depends on the selected back-end. Command line and embedded options allow users to select a back-end (Coq, HOL Light, or none), to set the internal precision used by MPFR bounds of intervals, to limit the depth of dichotomy splits, and to enable or disable warning messages.

2.1 Formalizing a problem

The logical formula that Gappa is expected to prove is written between brackets $\{ \}$ as presented below and it may contain any implication (\rightarrow), disjunction (\vee), conjunction (\wedge) of enclosures of mathematical expressions. Enclosures are either bounded ranges (**in**) or inequalities (\leq or \geq). Any identifier without definition is assumed to be universally quantified over the set of real numbers the first time Gappa encounters it.

```
{ x - 2 in [-2,0] /\ (x + 1 in [0,2] -> y in [3,4])
  -> not x <= 1 /\ x + y in ? }
```

The logical formula is first modified and loosely broken according to the rules of sequent calculus as presented below. Each of the new formulas is then verified by Gappa. Some ranges on the right of these sub-formulas can be left unspecified. Gappa then tries to suggest ranges where the logical formula is verified.

$$\begin{aligned} x \leq 1 \wedge x - 2 \in [-2, 0] &\implies x + 1 \in [0, 2] \vee x + y \in ? \\ x \leq 1 \wedge x - 2 \in [-2, 0] \wedge y \in [3, 4] &\implies x + y \in ? \end{aligned}$$

In order to be useful in the proof of the whole formula, the second sub-formula requires the first one to hold true. If Gappa cannot verify the first sub-formula, it will skip the verification of the second one.

Inequalities can be present on both sides of a sub-formula. On the left side, each inequality will be used only if Gappa is already able to compute an enclosure of the expression by some other means. On the right side, Gappa will introduce a reverted copy of the inequalities on the left side in order to increase the number of available hypotheses, as allowed by classical logic.

When proving a disjunction in a sub-formula, one of the sub-terms of the disjunction has to continuously hold with respect to the set of hypotheses. If Gappa cannot prove that the same sub-term always hold, it will be unable to prove that the whole disjunction holds.

Gappa produces an error message if an interval is written with reversed bounds or is so tight that Gappa needs to replace it with an empty interval. For example, the goal `1.3 in [1.3, 1.3]` can not be verified by Gappa, as the empty set is the biggest representable subset of the set `{1.3}`.

2.2 Definitions of rounded expressions and aliases

Typing large expressions in the logical formula would not be practical for the certification of software. Aliases to mathematical expressions are defined by constructions of the form `name = term` and `name` becomes available for later definitions, the logical formula and hints. It is neither an equality nor an affectation in any operational semantic but rather an alias. Gappa uses the defined aliases for its outputs and in the formal proof instead of machine generated names. A symbol cannot be defined more than once, even if the right hand sides of both definitions are equivalent. Neither can it be defined after having been used as an unbound variable. For example `b = a * 2; a = 1;` is not accepted by Gappa.

Gappa is specially designed to verify properties that may appear when certifying numerical codes. Rounding operators are used in the arithmetic expressions of these properties. They are real functions yielding rounded values according to the target data format (`precision`, `minimum_exponent`, and `lsb_weight`) and a predefined rounding mode amongst the ones presented Table 1. Floating- and fixed-point rounding operators can be expressed with the following operators:

```
float< precision, minimum_exponent, rounding_direction >(...)
fixed< lsb_weight, rounding_direction >(...)
```

The syntax above can be abbreviated for the floating-point formats of Table 2 and for (fixed-point) integer arithmetic:

```
float< name, rounding_direction >(...)
int< rounding_direction >(...)
```

The example below shows various ways of expressing rounded operations. Aliases are permitted for rounding operators and Line 1 defines `rnd` as rounding to nearest using IEEE 754 standard for 32 bit floating-point data [23]. When all the arithmetic operations on the right side of a definition are followed by the same rounding operator (as visible Line 2), this operator can be put once and for all at the left of the equal symbol (as presented Line 3). On this example, Gappa even complains that `y` and `z` are two different names for the same expression.

```
1 @rnd = float< ieee_32, ne>;
2 y = rnd(x * rnd(1 - x));
3 z rnd= x * (1 - x);
```

Table 1. Rounding modes available in Gappa. For modes that are not defined by IEEE 754 standard [23] and its forthcoming revision, readers are invited to review [24,25] and references herein.

Alias	Meaning
<code>zr</code>	toward zero
<code>aw</code>	away from zero
<code>dn</code>	toward minus infinity (down)
<code>up</code>	toward plus infinity
<code>od</code>	to odd mantissas
<code>ne</code>	to nearest, tie breaking to even mantissas
<code>no</code>	to nearest, tie breaking to odd mantissas
<code>nz</code>	to nearest, tie breaking toward zero
<code>na</code>	to nearest, tie breaking away from zero
<code>nd</code>	to nearest, tie breaking toward minus infinity
<code>nu</code>	to nearest, tie breaking toward plus infinity

Table 2. Predefined floating-point formats available in Gappa

Alias	Meaning
<code>ieee_32</code>	IEEE-754 single precision
<code>ieee_64</code>	IEEE-754 double precision
<code>ieee_128</code>	IEEE-754 quadruple precision
<code>x86_80</code>	extended precision on x86-like processors

Most truncated hardware operators [26] and some compound operators cannot be described as if they were computed to infinite precision and then rounded.

For such operators we revert to under-specified functions that produce results with a known bound on the relative error.

```
{add|sub|mul}_rel < precision [, minimum_exponent] >(..., ...)
```

If a minimum exponent is provided, Gappa does not instantiate any assumption that involves a result with an exponent below the minimum exponent. Otherwise, the error bound always hold and the absolute error is 0 when the result is 0.

2.3 Rewriting expressions to suppress some dependency effects

Let *accur* be an expression and *approx* an approximation of *accur* due to round-off errors, for example. The absolute error is $approx - accur$ and the relative error is $(approx - accur)/accur$. As soon as Gappa has computed ranges for *approx* and *accur*, it applies some theorems about interval subtraction and division to obtain some ranges for these errors.

Unfortunately, expressions *approx* and *accur* are strongly correlated and error ranges computed that way are useless. To suppress some dependency effects, Gappa manipulates error expressions through a set of standard pattern-matching and user-defined rewriting rules to reproduce many of the techniques used in numerical analysis and in computer arithmetic [27,8,28,29].

Standard rules kick in when the expressions of *approx* and *accur* are similar, e.g. $accur = a + b$ and $approx = \circ(c + d)$. Gappa rewrites the absolute error $approx - accur$ as $(\circ(c + d) - (c + d)) + ((c + d) - (a + b))$. It finds an enclosure of the left hand side by a theorem on the \circ rounding operator. For the right hand side, Gappa performs a second rewrite: $(c + d) - (a + b)$ is equal to $(c - a) + (d - b)$. This rewriting rule gives sensible results, if c and d are close to a and b respectively.

The first rule, $\circ(x) - y = (\circ(x) - x) + (x - y)$, has been applied by Gappa, because it knows that $\circ(x)$ is an approximation of x when \circ is a rounding operator. Gappa creates such a pair for any absolute or relative error that appears as a hypothesis of a logical sub-formula. Many rules operate on these pairs. For example, Gappa automatically replaces B by $b + -(b - B)$, if b and B pair as approximations.

Users may define other pairs with the following syntax $x \sim y$ that states as below that x is an approximation of y . Such pairs, additional rewriting rules, and directives of bisection, appear in the last section of the script for Gappa. When given the following script, Gappa warns the user that it already guessed the two hints and proposes some accurate bounds.

```
@floor = int<dn>;
{ x - y in [-0.1,0.1] -> floor(x) - y in ? }
floor(x) ~ x;
x ~ y;
```

Many rewriting rules are implemented in Gappa and they are sufficient to verify most properties on straight numerical applications. For intricate developments, users can add new rules to express some mathematical properties of their code. The rule `primary -> secondary` explicitly states that Gappa can use an

enclosure of **secondary** expression whenever it needs an enclosure of **primary** expression. Such rules usually explicit some techniques applied by designers that are no longer clear when readying the source code. We cannot expect an automatic tool to re-discover innovative techniques. Yet, we will incorporate in Gappa any technique that becomes commonly used.

In order for the previous rule to be valid, any value of **primary** must be contained in the computed enclosure of **secondary**. This property generally holds true if both expressions are equal. For example, Newton relation for the reciprocal can be written $x * (2 - x * y) - 1/y \rightarrow (x - 1/y) * (x - 1/y) * -y$. Any additional rule produces an hypothesis in the generated Coq file that must be guaranteed independently.

To detect mistypings early, Gappa tries to check if expressions are equal and warn if they are not. Note that Gappa does not check if divisors are always different from zero before applying user-defined rewriting rules. Yet, Gappa detects divisors that are trivially equal to zero in expressions that appear in rewriting rules. For example, $y \rightarrow y * (x - x) / (x - x)$ is most certainly an error.

As it discovers alternate expressions for one quantity, Gappa tries to enhance its bounds on the quantity by evaluating the new expressions. Tightening bounds on one quantity may allow to tighten bounds on quantities based on it. Gappa explores the graph of quantities breadth-first until the logical formula is proved or no range evolves anymore.

2.4 Subpaving the range of some quantities by bisection

The last kind of hint that can be used when Gappa is unable to prove a formula is to pave the range of some quantities and to prove independent results on each tile. Rewriting expressions is usually very efficient but it fails if different proof structures are needed on various parts of the range, as in the following example. Gappa cannot use the fact that $\text{rnd}(y) - y$ is always zero when $\frac{1}{2} \leq x \leq 3$, unless the last line is provided.

```
@rnd = float< ieee_32 , ne >;
x = rnd(x_);
y = x - 1;
z = x * (rnd(y) - y);
{ x in [0,3] -> |z| <= 1b-26 }
|z| $ x;
```

There are three constructions for bisection each involving a \$ sign in the hints section:

- Evenly split the range into as many sub-intervals as asked. E.g. `$ x in 6` splits the range of `x` in six sub-intervals. If the number of intervals is omitted (e.g. `$ x`) and no expression is present on the left of \$, the default is 4.
- Split an interval along user-provided points. E.g. `$ x in (0.5,2)` splits the range `[0,3]` of `x` in three sub-intervals, the middle one being `[0.5,2]`.
- The third kind of bisection tries to find by dichotomy a good subpaving such that a goal of the logical formula holds true. This requires the range of this

goal to be specified in the logical proposition, and the enclosed expression has to be indicated on the left of the `$` symbol.

More than one bisection hint can be used. And hints of the third kind can try to satisfy more than one goal at once. The two hints below will be used sequentially one after the other. The first one will split the range of u until all the enclosures on a , b , and c are verified.

```
a, b, c $ u;
d, e $ v;
```

Users may build higher dimension subpavings by using more than one term on the right of the `$` symbol, reaching quickly combinatorial explosions though. The following hint asks Gappa to find a set of sub-ranges of u and w such that the goals on a and b are satisfied when the range of v is split into three sub-intervals.

```
a, b $ u, v in 3, w
```

3 Handling automatic proof checkers

The generated Coq script contains the following lemma whenever the certificate relies on interval addition to prove a proposition, e.g. “if $x \in [1, 2]$ (property `p1`) and $y \in [3, 4]$ (property `p2`), then $x + y \in [0, 6]$ (property `p3`)”.

```
1 Lemma l1 : p1 -> p2 -> p3.
2   intros h0 h1.
3   apply add with (1 := h0) (2 := h1) ; finalize.
4   Qed.
```

The first line defines the lemma: if the hypotheses `p1` and `p2` are verified, then the property `p3` is true too. The second line starts the proof in a suitable state by using the `intros` tactic of Coq. The third line applies the `add` theorem of Gappa support library with the `apply` tactic.

The `add` theorem is as follows. `lower` and `upper` are functions that return the lower and the upper bound of an interval of type `FF` represented by a pair of dyadic fractions. `Fplus2` is the addition of dyadic fractions. `Fle2` compares two dyadic fractions (less or equal) and returns a boolean. The `BND` predicate holds, when its first argument, an expression on real numbers, is an element of its second argument, an interval defined by dyadic fraction bounds (`IFF`).

```
Definition add_helper (xi yi zi : FF) :=
  Fle2 (lower zi) (Fplus2 (lower xi) (lower yi)) &&
  Fle2 (Fplus2 (upper xi) (upper yi)) (upper zi).
```

```
Theorem add :
  forall x y : R, forall xi yi zi : FF,
  BND x xi -> BND y yi ->
  add_helper xi yi zi = true ->
  BND (x + y) zi.
```


The mathematical expression of the theorem is as follows:

$$\begin{aligned} \text{add} : \forall x, y \in \mathbb{R}, \quad \forall I_x, I_y, I_z \in \mathbb{IF}, \\ x \in I_x \Rightarrow y \in I_y \Rightarrow \\ f_{\text{add}}(I_x, I_y, I_z) = \text{true} \Rightarrow \\ x + y \in I_z. \end{aligned}$$

If we simply needed a theorem describing the addition in interval arithmetic, the $f_{\text{add}}(I_x, I_y, I_z) = \text{true}$ hypothesis would be replaced by $I_x + I_y \subseteq I_z$. But we also need for the theorem hypotheses to be automatically checkable. It is the case for the $x \in I_x$ and $y \in I_y$ hypotheses of the **add** theorem, since they can be directly matched to the hypotheses **h0** ($x \in [1, 2]$) and **h1** ($y \in [3, 4]$) of lemma 11.

Hypothesis $I_x + I_y \subseteq I_z$, however, cannot be matched so easily. Consequently it is replaced by an equivalent boolean expression that can be computed by a proof checker. In lemma 11, the computation is triggered by the **finalize** tactic that checks that the current goal can be reduced to $\text{true} = \text{true}$. This concludes the proof.

All the theorems of Gappa companion library are built the same way: instead of having standard hypotheses that Coq would be unable to automatically decide, they have a computable boolean expression. When this expression evaluates to *true*, the standard hypotheses are proved to be true, and the goal of the theorem applies. This approach is a simpler form of reflection techniques [30]. Although the use of booleans seems to restrict it to the Coq proof checker, the interval arithmetic library [13] developed for PVS shows that proofs through interval computations are also attainable to other proof assistants.

Ensuring these computable boolean expressions exist is the reason why all the interval bounds are dyadic fractions ($m \cdot 2^n$ with m and n relative integers). Such numbers can easily and efficiently be added, multiplied, and compared. Rational numbers could also have been used. They would have been almost as efficient and would have provided a division operator. But common floating-point properties involved in certifying numerical code are better described and verified by using dyadic fractions.

Although enclosure (**BND**) is the only predicate available to users, Gappa internally relies on more predicates to describe properties on an expression x . In particular, the **FIX** and **FLT** predicates allow to express that the set of computer numbers is generally a discrete subset of the real numbers, while intervals only consider connected subsets. These predicates will appear in intermediate lemmas of the generated certificates.

$$\begin{aligned} \text{BND}(x, [a, b]) &\equiv a \leq x \leq b \\ \text{ABS}(x, [a, b]) &\equiv 0 \leq a \leq |x| \leq b \\ \text{FIX}(x, e) &\equiv \exists m \in \mathbb{Z}, x = m \cdot 2^e \\ \text{FLT}(x, p) &\equiv \exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge |m| < 2^p \end{aligned}$$

4 Perspectives and concluding remarks

In our approach to program certification, proof generation and proof verification are two distinct steps. The first one is done by Gappa with its own computational methods, and the second one is done by a proof checker with the help of a support library. The proof checker never has to compute any interval, it just checks that the intervals generated by Gappa make the boolean expressions evaluate to *true*, and hence are valid. In particular, there is absolutely no need for Gappa to compute the best enclosing interval of an expression. As long as the proof remains correct, any interval can be used.

Consequently, an interval can be widened if it does not impact the final result. For example, manipulating the expression $x/\sqrt{2}$ will sooner or later require $\sqrt{2} \neq 0$ to be proved. This is done by computing an enclosing interval of $\sqrt{2}$ and verifying that its lower bound is positive. Hence there is no need to compute an enclosing interval with thousands of bits of precision, the interval $[1, 2]$ is accurate enough.

Such simplifications are important, since a proof checker like Coq is considerably slower than a specialized mathematical library. It is especially true for case studies: searching for a better subpaving and certifying it, will always be faster than directly certifying the first subpaving that has been found by Gappa. The time spent in doing all the computations over and over in order to find a better subpaving is negligible in comparison to the time necessary to certify the property on one single tile with the proof checker.

The whole work of generating the proof is pushed toward the external program. All the intervals are precomputed and none of the complex tactics of Coq are used. The proof checker only has to be able to add, multiply, and compare integers; it does not have to be able to manipulate rational or real numbers. If it was not for the readability of the proof, the tool could directly generate the lambda-term describing the proof, and Coq would just have to compute its type. Consequently, one of our goal is to generate proofs not only for Coq, but for other proof checkers too.

Branches and loops handling are outside the scope of this work. Both problems are not new to program verification and nice results have been published in both areas. We do not want to propose our solution for these problems. Our decision is to interact with the two following tools.

- Why [31] is a tool to certify programs written in a generic language (C and Java can be converted to this language). It certifies appropriate memory allocation and usage. It is able to handle hierarchically structured code with functions and assertions. Why also takes care of conditional branches. It duplicates the appropriate proofs and guarantees that both pieces of code meet their shared post-conditions. Used together, Why and Gappa will be able to handle complex numeric codes.
- Fluctuat [32,33] handles loops by effectively computing loop invariants. Once these invariants are provided, Gappa can certify the correct behavior of any numerical code. Results of Fluctuat will be used as oracles and certified

by Gappa. Should there be a significant bug in Fluctuat, Gappa will stop without being able to meet its goals as it cannot certify erroneous results.

The developments presented so far already allow us to guarantee the correct behavior of many useful functions. Our software, a user's guide including a grammar, a description of the example presented Figure 1 and links and details of some projects using Gappa are available on the Internet at the address below.

<http://lipforge.ens-lyon.fr/www/gappa/>

In particular, our tool is being used to certify CRLibm, a library of elementary functions with correct rounding in the four IEEE-754 rounding modes and performances comparable to standard mathematical libraries [2,29]. Gappa is also used to develop robust semi-static filters for the CGAL project [3].

```
@rnd = float< ieee_32, ne >;

a1 = 8388676b-24;
a2 = 11184876b-26;
l2 = 12566158b-48;
s1 = 8572288b-23;
s2 = 13833605b-44;

r2 rnd= -n * l2;
r rnd= r1 + r2;
q rnd= r * r * (a1 + r * a2);
p rnd= r1 + (r2 + q);
s rnd= s1 + s2;
e rnd= s1 + (s2 + s * p);

R = r1 + r2;
S = s1 + s2;

E = s1 + (s2 + S * (r1 + (r2 + R * R * (a1 + R * a2))));
Er = S * (1 + R + a1 * R * R + a2 * R * R * R + 0);
E0 = S0 * (1 + R0 + a1 * R0 * R0 + a2 * R0 * R0 * R0 + Z);

{ Z in [-55b-39,55b-39] /\ S - S0 in [-1b-41,1b-41] /\
  R - R0 in [-1b-34,1b-34] /\ R in [0,0.0217] /\ n in [-10176,10176] ->
  e in ? /\ e - E0 in ? }

e - E0 -> (e - E) + (Er - E0);
r1 -> R - r2;
```

Fig. 1. Gappa script for an implementation of an almost correctly rounded elementary function in single and double precision [34] later validated in HOL Light [35].

References

1. Revy, G.: Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Technical Report ensi-00119498, École Normale Supérieure de Lyon (2006)
2. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France (2006) 1318–1322
3. Melquiond, G., Pion, S.: Formally certified floating-point filters for homogeneous geometric predicates. Theoretical Informatics and Applications (2007) To appear.
4. Michard, R., Tisserand, A., Veyrat-Charvillon, N.: Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In: Symposium en Architecture de Machines, Perpignan, France (2006) 1318–1322
5. Information Management and Technology Division: Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office (1992)
6. Lions, J.L., et al.: Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France (1996)
7. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM (1997)
8. Higham, N.J.: Accuracy and stability of numerical algorithms. SIAM (2002) Second edition.
9. Meyer, B.: UML: the positive spin. American Programmer (1997)
10. Schlumberger: Schlumberger leads the way in smart card security with common criteria EAL7 security methodology. Press Releases (2003)
11. Rockwell Collins: Rockwell Collins receives MILS certification from NSA on microprocessor. Press Releases (2005)
12. Gameiro, M., Manolios, P.: Formally verifying an algorithm based on interval arithmetic for checking transversality. In: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas (2004) 17
13. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In Montuschi, P., Schwarz, E., eds.: Proceedings of the 17th Symposium on Computer Arithmetic, Cape Cod, Massachusetts (2005) 188–195
14. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE **91**(1) (2003) 29–39
15. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq proof assistant: a tutorial: version 8.0. (2004)
16. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer-Verlag (2004)
17. Neumaier, A.: Interval methods for systems of equations. Cambridge University Press (1990)
18. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied interval analysis. Springer (2001)
19. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation. Technical Report 05.12, Hamburg University of Technology, Hamburg, Germany (2005)
20. Daumas, M., Melquiond, G.: Generating formally certified bounds on values and round-off errors. In: Real Numbers and Computers, Dagstuhl, Germany (2004) 55–70
21. Brönnimann, H., Melquiond, G., Pion, S.: The Boost interval arithmetic library. In: Real Numbers and Computers, Lyon, France (2003) 65–80

22. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. Technical Report RR-5753, INRIA (2005)
23. Stevenson, D., et al.: An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices* **22**(2) (1987) 9–25
24. Even, G., Seidel, P.M.: A comparison of three rounding algorithms for IEEE floating-point multiplication. In Koren, I., Kornerup, P., eds.: *Proceedings of the 14th Symposium on Computer Arithmetic*, Adelaide, Australia (1999) 225–232
25. Boldo, S., Melquiond, G.: When double rounding is odd. In: *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, Paris, France (2005)
26. Texas Instruments: TMS320C3x — User's guide. (1997)
27. Kahan, W.: Further remarks on reducing truncation errors. *Communications of the ACM* **8**(1) (1965) 40
28. Boldo, S., Daumas, M.: A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms* **37**(1-4) (2004) 45–60
29. de Dinechin, F., Defour, D., Lauter, C.: Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research report 5137, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France (2004)
30. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*, London, United Kingdom (1997) 515–529
31. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud (2003)
32. Martel, M.: Propagation of roundoff errors in finite precision computations: a semantics approach. In: *11th European Symposium on Programming*, Grenoble, France (2002) 194–208
33. Putot, S., Goubault, E., Martel, M.: Static analysis based validation of floating point computations. In: *Novel Approaches to Verification*. Volume 2991 of *Lecture Notes in Computer Science*, Dagstuhl, Germany (2004) 306–313
34. Tang, P.T.P.: Table driven implementation of the exponential function in IEEE floating point arithmetic. *ACM Transactions on Mathematical Software* **15**(2) (1989) 144–157
35. Harrison, J.: Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory (1997)