Combining Coq and Gappa for Certifying Floating-Point Programs[⋆]

Sylvie Boldo[1,2], Jean-Christophe Filliâtre[2,1], and Guillaume Melquiond[1,2]

[1] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
[2] LRI, Université Paris-Sud, CNRS, Orsay, F-91405
Sylvie.Boldo@inria.fr
Jean-Christophe.Filliatre@lri.fr
Guillaume.Melquiond@inria.fr

Abstract. Formal verification of numerical programs is notoriously difficult. On the one hand, there exist automatic tools specialized in floating-point arithmetic, such as Gappa, but they target very restrictive logics. On the other hand, there are interactive theorem provers, such as Coq, that handle a general-purpose logic but that lack proof automation for floating-point properties. To alleviate these issues, we have implemented a mechanism for calling Gappa from a Coq interactive proof. This paper presents this combination and shows on several examples how this approach offers a significant speedup in the process of verifying floating-point programs.

1 Introduction

Numerical programs typically use floating-point arithmetic [1]. Due to their limited precision and range, floating-point numbers are only an approximation of real numbers. Each operation may introduce an inaccuracy and their total contribution is called rounding error. Moreover, some real operations may not be available as sequences of floating-point operations, *e.g.* infinite sums or integrals. This introduces another inaccuracy called method error. Both errors make it somehow complicated to know what floating-point programs actually compute with respect to the initial algorithms on real numbers.

One way to proceed is to give a program a precise specification of its accuracy. Generally speaking, a specification explains what can be expected from the result given facts about the inputs. Typically, it bounds the sum of both rounding and method errors. For example, the specification for a function `float_cos` defined on the `double` type of floating-point numbers may be the following:[3]

$$\forall x : \texttt{double}, \quad |x| \le 2\pi \ \Rightarrow \ \left|\frac{\cos(x) - \texttt{float\_cos}(x)}{\cos(x)}\right| \le 2^{-53}.$$

[⋆] This research was partially supported by projects ANR-05-BLAN-0281-04 "CerPAN" and ANR-08-BLAN-0246-01 "F∮ST".

[3] Note that $\pi/2$ cannot be represented by a floating-point number, therefore $\cos(x)$ cannot be zero.

Such an inequality is typically proved using pen and paper. It can be deduced from the specification of each floating-point operation and the mathematical properties of the cosine function.

Such proofs are notoriously difficult and error-prone. In order to increase confidence, one may use formal methods. For instance, one may verify the correctness of a program using the Coq proof assistant with a suitable formalization of floating-point arithmetic (Section 2.2). This method, however, makes the process even more tedious, due to the high number of explicit proof steps. Another approach is to use an automated theorem prover such as Gappa (Section 2.3), which is very efficient at proving bounds, for instance on rounding errors. But Gappa only tackles the floating-point fragment of a program, and thus may not always be able to complete the verification process.

To benefit from both approaches, we have implemented a mechanism for calling Gappa from a Coq interactive proof. As usual, the user issues tactics to split the goal into simpler subgoals. Examples of such tactics are logical cut, case analysis, or induction. Then Gappa can be called once a subgoal is specific enough to fit into its floating-point formalism. From a technical point of view, Gappa is called as an external prover. This does not weaken the confidence in Coq formal proofs since Gappa produces a proof trace that is checked by Coq (Section 4).

This combination of Coq and Gappa does not radically change the way to tackle rounding and method errors. It simply eases the use of traditional approaches in a formal setting. C programs illustrating this point are given in Section 3. Their specifications are written in the formalism of the Caduceus tool (Section 2.1), which generates Coq proof obligations expressing the program correctness. The combination of all these tools (Caduceus, Coq, Gappa) greatly simplifies the formal verification of these examples.

2 Context

This section introduces the three tools Caduceus, Coq and Gappa.

2.1 Verification of C Programs

The Why platform[4] is a set of tools for deductive verification of Java and C programs [2]. In this paper, we only focus on verification of C programs but the results would apply to Java programs as well. The verification of a given C program proceeds as follows. First, the user specifies requirements as annotations in the source code, in a special style of comments. Then, the annotated program is fed to the tool Caduceus [3], which is part of the Why platform, and verification conditions (VCs for short) are produced. These are logical formulas whose validity implies the soundness of the code with respect to the given specification. Finally, the VCs are discharged using one or several theorem provers,

[4] The Why platform is free software available at `http://why.lri.fr/`.

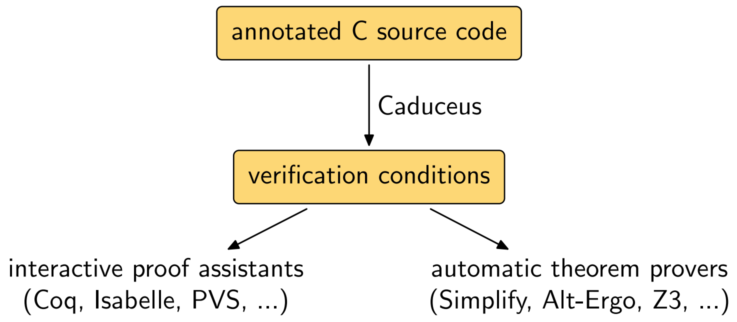
annotated C source code
Caduceus
verification conditions
interactive proof assistants
(Coq, Isabelle, PVS, ...)
automatic theorem provers
(Simplify, Alt-Ergo, Z3, ...)

Fig. 1. The Caduceus tool.

which range from interactive proof assistants such as Coq to purely automatic theorem provers such as Alt-Ergo [4]. The workflow is illustrated on Figure 1.

Annotations are inserted in C source code using comments with a leading `@`. They are written in first-order logic and re-use the syntax of side-effect free C expressions. For instance, here is a code excerpt where an array `t` is searched for a zero value.

```
//@ invariant 0 <= i
for (i = 0; i < n; i++) {
  if (t[i] == 0) break;
}
//@ assert i < n => t[i] == 0
```

The `for` loop is given a loop invariant, as in traditional Hoare logic [5]. (In that case, the invariant could be found automatically.) A loop invariant typically generates two VCs: one to show that it holds right before the loop is entered; and one to show that it is preserved by the loop body. In this example, an assertion is also manually inserted right after the loop, which results in a VC for this program point. Additional VCs are produced to establish the safe execution of the code, *i.e.* that the program does not perform any division by zero or any array access out of bounds. In this example, a VC requires to show that `t[i]` is a legal array access, which may or may not be provable depending on hypotheses regarding `t` and `n`.

Verification with Caduceus is modular: Each function is given a *contract* and proved correct w.r.t. the contracts of the functions it calls.[5] For instance, a partial contract for a function sorting an array of integer could be

```
/*@ requires
  @   0 <= n && \valid_range(t, 0, n-1)
  @ assigns
  @   t[0..n-1]
  @ ensures
  @   \forall int i,j; 0 <= i <= j < n => t[i] <= t[j] */
void sort(int *t, int n);
```

[5] That means we only establish *partial correctness* of recursive functions.

The contract contains three clauses. Keyword `requires` introduces a precondition, that is a property assumed by the function and proved at the caller site. In this example, it states that `n` is nonnegative and that all indices from 0 to `n-1` in `t` can be safely accessed. Conversely, keyword `ensures` introduces a postcondition, that is a property provided by the function, right before it returns. Here it states that the array is sorted in increasing order.[6] Finally, keyword `assigns` introduces the memory locations possibly modified by the function, which means that any other memory location is left unchanged by a call to this function. Here, it states that only the array elements `t[`i`]` for $0 \leq i <$ `n` are possibly assigned.

Caduceus handles a large fragment of ANSI C, with the notable exception of pointer casts and unions. It does handle floating-point arithmetic, using a model where each floating-point number is seen as a triple of real numbers [6]. The first component is the floating-point number itself, as it is computed. The second component is the real number that would have been computed if roundings were not performed. The third component is a ghost variable attached to the floating-point number and which represents the ideal value that the programmer intended to compute. Annotations are written using real numbers only, and the three components of a floating-point variable `x` can be referred to within annotations: `x` itself stands for the first component; `\exact(x)` for the second one; and `\model(x)` for the third one. Thus the user can refer to the rounding error as the difference between the first two, and to method error as the difference between the last two. Examples are given in Section 3.

Since the general-purpose automatic provers do not support this model of floating-point arithmetic, we have formalized it in the Coq proof assistant.

2.2 The Coq Proof Assistant

The Coq proof checker [7,8] is a proof assistant based on higher-order logic. One may express properties such as "there exists a function which has such and such properties" or "every relation that verifies such hypothesis has a certain property" and check proofs about these. Proofs are built using tactics (such as applying a theorem, rewriting, computing, etc.). A Coq file contains the stating of lemmas and their proofs as a sequence of tactics in the Coq language.

The Coq standard library contains an axiomatization of real numbers [9]. Few automation is provided to reason about real numbers. As a consequence, the proof of a typical lemma such as $0 < 1 - 2^{-52}$ is already a few lines long:

```
Lemma OneMinusUlpPos: (0 < 1 - powerRZ 2 (-52))%R.
Proof.
  apply Rlt_Rminus.
  unfold powerRZ.
  rewrite <- Rinv_1 at 3.
  apply Rinv_1_lt_contravar ; auto with real.
Qed.
```

[6] For the specification to be complete, the postcondition should also state that the array is a permutation of its initial value. It can be done, but is omitted here for the sake of simplicity.

The proof is done backward, by transforming the conclusion until it trivially derives from the hypotheses. This proof starts by applying the theorem `Rlt_Rminus` (line 3) since $0 < 1 - 2^{-52}$ is a consequence of $2^{-52} < 1$. The definition of `powerRZ` is then unfolded (line 4) so that 2^{-52} is converted to $(2^{52})^{-1}$. We replace 1 by 1^{-1} (theorem `Rinv_1`, line 5). At this point, the goal has become $(2^{52})^{-1} < 1^{-1}$. After applying theorem `Rinv_1_lt_contravar` (line 6), the remaining goals are $1 < 2^{52}$ and $1 \leq 1$, which are solved automatically by the tactic `auto` (line 6).

A high-level formalization of floating-point arithmetic [10,11] is also available in Coq. A floating-point number is a pair of integers (m, e) which represents the real number $m \times 2^e$. The value of the mantissa m and the exponent e are bounded according to the floating-point format. For example, in IEEE-754 double-precision format [1], the pair verifies $|m| < 2^{53}$ and $-1074 \leq e$. This library[7] contains a large number of floating-point definitions and theorems and has been used to prove many old and new properties [12].

Most floating-point proofs rely on computations on real numbers, such as deciding $0 < 1 - 2^{-52}$ or bounding method error. Such goals can be addressed using the `interval` tactic [13]. This reflexive tactic, based on interval arithmetic, decides inequalities by bounding real expressions thanks to guaranteed floating-point arithmetic. Once done with method error, the user is left with VCs related to rounding errors, which Gappa is typically designed for.

2.3 The Gappa Tool

Gappa[8] is a tool dedicated to proving arithmetic properties on numerical programs [14,15]. Given a logical proposition expressing bounds on mathematical real-valued expressions, Gappa checks it holds. The following is such a proposition and below is its transcription in Gappa's input language.

$$\forall x, y \in \mathbb{R}, \quad |x| \leq 2 \wedge y \in [1,9] \Rightarrow x \times x + \sqrt{y} \in [1,7]$$

```
{ |x| <= 2 /\ y in [1,9] -> x * x + sqrt(y) in [1,7] }
```

In order to verify the proposition, Gappa first analyzes which expressions may be of interest. Then it tries to enclose them in intervals by performing a saturation over its library of theorems on interval arithmetic, forward error analysis, and algebraical identities. Gappa stops when it reaches enclosures small enough to be compatible with the right-hand side of the proposition or when the saturation does no longer improve the enclosures.

Once Gappa has verified the proposition, it generates a formal proof. To increase confidence, this proof script can then be mechanically checked by an independent proof system, such as Coq or HOL light.

If Gappa fails to prove the proposition, the user can suggest the tool to perform a bisection — splitting input intervals until the proposition holds on

[7] Available at `http://lipforge.ens-lyon.fr/www/pff/`.
[8] Available at `http://lipforge.ens-lyon.fr/www/gappa/`.

each sub-intervals — or augment the library of theorems with new mathematical identities. Gappa will then assume these equalities hold; they will appear as hypotheses of the generated formal proof.

Expressing Floating-Point Programs. In addition to universally-quantified variables on $\mathbb{R}$, basic arithmetic operators ($+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$), and numerical constants, Gappa expressions can also contain *rounding* operators. The integer-part functions, $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$, are instances of such operators. Since the IEEE-754 standard [1] mandates that "a floating-point operator shall behave as if it was first computing the *infinitely-precise* value and then *rounding* it so that it fits in the destination floating-point format", having appropriate rounding operators is sufficient to express the computations of a floating-point program.

The following script is similar to the previous one, but all the expressions are now as if they had been computed in single precision with rounding to nearest (tie-breaking to even mantissa).

```
@rnd = float<ieee_32,ne>;
z = rnd(rnd(x * x) + rnd(sqrt(y)));
{ |x| <= 2 /\ y in [1,9] -> z in [1,7] }
```

Note that Gappa only manipulates expressions on real numbers. As a consequence, infinities and NaNs (Not-a-Numbers) are no part of this formalism: Rounding operators return a real value and there is no upper bound on the magnitude of the input numbers. This means that NaNs and infinities will not be generated nor propagated as they would in IEEE-754 arithmetic. Nonetheless, one may still use Gappa to prove propositions related to these special values, or rather their absence.

For instance, if one proves that a Gappa-rounded value is smaller than the biggest floating-point number in the working format, then the actual IEEE-754 computation is guaranteed not to overflow, by definition of overflow. Therefore, in order to check that computations in the example above are overflow-safe, one can run Gappa on the following script:[9]

```
@rnd = float<ieee_32,ne>;
z = rnd(rnd(x * x) + rnd(sqrt(y)));
{ |x| <= 2 /\ y in [1,9]
  -> z in [1,7] /\ |rnd(x * x)| <= 0x1.FFFFFEp127 /\
     |rnd(sqrt(y))| <= 0x1.FFFFFEp127 }
```

Verifying Accuracy. While examples above show that Gappa can bound ranges of floating-point variables, this is only a small part of its purpose. This tool was designed to prove bounds on computation errors, which also happen to be real-valued expressions. Let us assume that the developer actually needed the infinitely-precise result $M_z = x^2 + \sqrt{y}$. Is the computed result z sufficiently

[9] The number `0x1.FFFFFEp127` is the C99-like representation of the biggest finite floating-point numbers for IEEE-754 single-precision format.

close to this ideal value M_z? This can be answered by bounding the absolute error $z - M_z$:[10]

```
@rnd = float<ieee_32,ne>;
Mz = x * x + sqrt(y);
z = rnd(rnd(x * x) + rnd(sqrt(y)));
{ |x| <= 2 /\ y in [1,9] -> |z - Mz| <= 1b-21 }
```

For the sake of simplicity, M_z has the same operations without rounding than z in the example above. This is not a requirement, as Gappa is also able to bound errors when M_z is a completely different expression. Note also that Gappa is not limited to absolute errors; it can handle relative errors in a similar way, which is especially important when proving floating-point properties.

3 Proving Floating-Point Programs

Before describing the inner working of the Coq-Gappa combination, we illustrate its use on the verification of three typical floating-point programs.

3.1 Naive Cosine Computation

The first example is an implementation of the cosine function for single-precision floating-point arithmetic. To present a complete Coq proof, we have simplified the function by removing its argument-reduction step. Thus, input x is required to have already been reduced to a value close to zero; only the polynomial evaluation has to be performed. The specification of the function states that, for $|x|$ smaller than 2^{-5}, the computed value `\result` is equal to $\cos(x)$ up to 2^{-23}.

```
/*@ requires |x| <= 1./32
  @ ensures  |\result - cos(x)| <=  2^^(-23)
  @ */
float toy_cos(float x) {
  return 1.f - x * x * .5f;
}
```

Note that 2^{-23} is a tight bound on the error of this function. It ensures that the computed result is one of the floating-point numbers close to the mathematical value $\cos(x)$.

Given the annotated C code above, Caduceus generates a VC stating the accuracy of the result, which can be formally proved with the Coq script below.[11]

[10] The number `1b-21` means 2^{-21}, which is almost the optimal upper bound on the specified absolute error.

[11] The Coq script is reproduced *verbatim*. In particular, some terms are obfuscated due to Coq renaming them to prevent conflicts. So `f` designates in fact the variable x; and `Rtrigo_def.cos` is the name of the cosine function in Coq's standard library.

```
1 Proof.
2   intros; why2gappa; unfold cos.
3   assert ( Rabs ((1 - (f*f) * (5/10)) - Rtrigo_def.cos f)
4               <= 7/134217728 )%R
5     by interval with (i_bisect_diff f).
6   gappa.
7 Qed.
```

The first part of the proof script (line 2) turns the goal into a user-friendly form: the `why2gappa` tactic cleans the goal by expanding and rewriting some Caduceus-specific notations. At this point, assuming that $\circ(\cdot)$ is the rounding operation from a real number to the nearest single-precision floating-point number, the user has to prove the following goal:

$$\begin{aligned}&\forall x : \texttt{float}, \quad |x| \leq \tfrac{1}{32} \Rightarrow \\ &\qquad |\circ(\circ(1) - \circ(\circ(x \times x) \times \circ(5/10))) - \cos(x)| \leq 2^{-23}.\end{aligned}$$

As would be done with a pen-and-paper verification, the formal proof of this goal starts by computing and proving a bound on the method error. Since this bound influences the choice of the polynomial, it is performed before the algorithm is written. For this example, we chose a degree-2 Taylor approximation because a computer algebra system told us the method error would be smaller than 7×2^{-27}. So we are asserting this property in Coq (lines 3 and 4) and we prove it (line 5). The assertion is proved by the `interval` tactic [13]. Its option `i_bisect_diff` tells the tactic to recursively perform a bisection on the interval enclosure $[-2^{-5}, 2^{-5}]$ of x, until a first-order interval evaluation of the method error $(1 - (x \times x) \times (5/10)) - \cos(x)$ gives a compatible bound on all the sub-intervals.

Once the assertion is proved and hence available as an hypothesis, the user has to prove the following property:

$$\begin{aligned}&\forall x : \texttt{float}, \quad |x| \leq \tfrac{1}{32} \Rightarrow \\ &\qquad |(1 - (x \times x) \times (5/10)) - \cos(x)| \leq 7 \cdot 2^{-27} \Rightarrow \\ &\qquad |\circ(\circ(1) - \circ(\circ(x \times x) \times \circ(5/10))) - \cos(x)| \leq 2^{-23}.\end{aligned}$$

This is achieved by the `gappa` tactic (line 6). It calls Gappa and then uses the Coq script the tool generates in order to finish the proof. Note that the Gappa tool takes advantage of the inequality proved by the `interval` tactic as it knows nothing about the cosine.

3.2 Discretization of a Partial Differential Equation

The second example is a numerical code about acoustic waves by F. Clément [16]. Given a rope attached at its two ends, a force initiates a wave, which then undulates according to the following mathematical equation:

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = 0.$$

This equation can be discretized in both space and time. At a given abscissa i and a given time k, the value `p[i][k]` is the position of the rope. Value i ranges from 0 to n_i and value k ranges from 0 to n_k. The main loop of the program is annotated as follows:

```
/*@ invariant 1 <= k <= nk
        && analytic_error(p,ni,ni,k,a) */
for (k=1; k<nk; k++) {
  p[0][k+1] = 0.;

  /*@ invariant 1 <= i <= ni
          && analytic_error(p,ni,i-1,k+1,a) */
  for (i=1; i<ni; i++) {
    dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
    p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
  }

  p[ni][k+1] = 0.;
}
```

where `a` is an approximation of an exact constant A derived from c, n_i and n_k.

The predicate `analytic_error` states the exact analytical expression of the rounding error. It also states that the rounding error of a single iteration is smaller than a known value. More precisely, it bounds the absolute value of

$$\varepsilon_i^{k+1} := p_i^{k+1} - (2p_i^k - p_i^{k-1} + A \times (p_{i+1}^k - 2p_i^k + p_{i-1}^k)).$$

Under some hypotheses on A and the ranges of (p_i^k), we could prove that $|\varepsilon_i^{k+1}| \leq 85 \times 2^{-52}$ (and a similar property concerning the initialization of the p_i^1). The original Coq proof amounts to 735 lines of tactics. Thanks to the `gappa` tactic, we were able to

– improve the result: we now have the formal proof that $|\varepsilon_i^{k+1}| \leq 80 \times 2^{-52}$;
– drastically cut off the size of the proof script: the 735 lines of tactics reduce to 10.

This is a tremendous improvement. Not only is the new proof script dramatically shorter and simpler to write, but it is also more amenable to future changes and maintenance. Indeed, if the program is to be modified in such a way that the error slightly increases, the initial proof would be completely broken and only a small part could be re-used. Using Gappa, the situation is different: While the statement of the theorem would change, the proof would probably be robust enough to remain valid.

3.3 Preventing Overflows

Another type of proof that greatly benefits from automation is overflow proofs. Typically, one wants to guarantee that no overflows happen. To do so, it is usually sufficient to bound the program inputs. The resulting VCs are especially tedious to prove. As a consequence, the bounds are often over-estimated in order to simplify the demonstrations. This is the case for the following example. This program computes an accurate discriminant using Kahan's algorithm [17].

```
/*@ requires xy==round(x*y) &&
  @    (x*y==0 || 2^^(-969) <= |x*y|) &&
  @    |x| <= 2^^995 && |y| <= 2^^995 && |x*y| <= 2^^1022
  @ ensures \result==x*y-xy
  @ */
double exactmult(double x, double y, double xy);

/*@ requires
  @      (b==0    || 2^^(-916) <= |b*b|) &&
  @      (a*c==0  || 2^^(-916) <= |a*c|) &&
  @      |b| <= 2^^510 && |a| <= 2^^995 && |c| <= 2^^995 &&
  @      |a*c| <= 2^^1021
  @ ensures \result==0 ||
  @      |\result-(b*b-a*c)| <= 2*ulp(\result)
  @ */

double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=exactmult(b,b,p);
    dq=exactmult(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

The formal proofs for this program (including overflows) have been done in [18,19]. Here we only focus on the overflows of the discriminant; we do not care about the `exactmult` function.

All the overflow proofs were first done prior to the `gappa` tactic. For seven proof obligations, it took more than 420 lines of Coq. Using the tactic, the proofs reduce to 35 lines (about 5 lines per theorem). The Coq compilation time, however, is about 5 times greater. Nevertheless, the profit is clear in the verification process as the time for developing the proof overwhelms the time for compiling it.

It is also interesting to note that the specification is also improved. The hypothesis $|a \times c| \leq 2^{1020}$ in the original proof [19] was too strong; we proved instead that $|a \times c| \leq 2^{1021}$ is sufficient to guarantee that no overflows occur. The proof was not modified at all after changing the annotations. This means that the automation is sufficient to use exactly the same proof when modifying slightly the specification. This is really worthwhile for proof maintenance.

4 Implementation Details

The `gappa` tactic is built from two components. The tactic itself is available from the standard V8.2 Coq distribution. Gappa is an external stand-alone tool, which comes with its own library of Coq theorems.

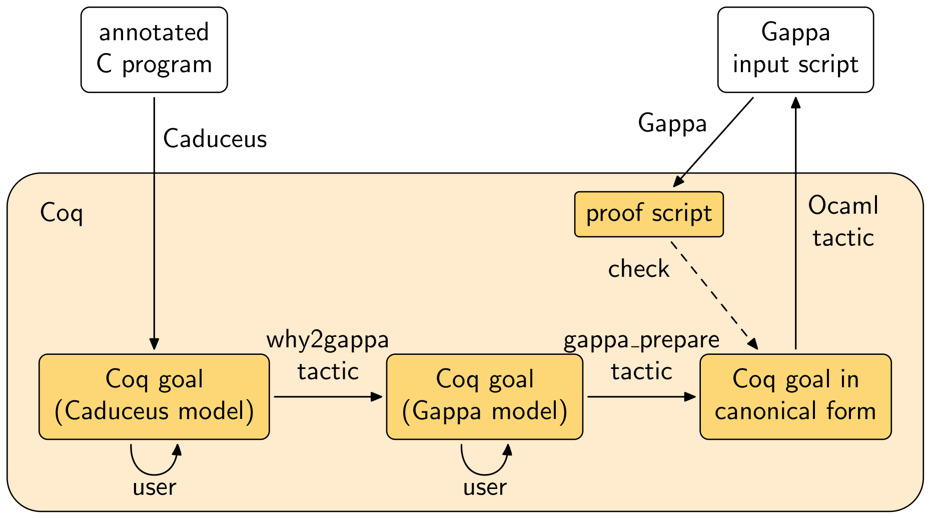

Fig. 2. Dataflow in the Coq and Gappa combination

Figure 2 describes the process of performing a formal certification of a C program using Coq and Gappa. Starting with an annotated C program, Caduceus generates VCs corresponding to the specification of this C code. Lots of these proof obligations can be discharged by automatic provers. The most complicated ones, especially those involving floating-point properties, are left to the user. The Caduceus tool therefore generates template Coq scripts, and the user has to fill in the blanks.

At this point, the Coq goals are expressed in the floating-point model of Caduceus. As usual with a proof assistant, the user issues tactics in order to split the goal into simpler subgoals that can be handled automatically. If one subgoal is in the scope of the Gappa tool, the user can proceed as follows.

First of all, the goal has to be translated to the floating-point model of Gappa. Stored in an auxiliary library, some theorems state that both models are equivalent. In particular, if a floating-point number is the closest to a real number in the Caduceus model, then it is the result of a rounding function in the Gappa model, and reciprocally. The `why2gappa` tactic automatically applies these theorems to rewrite the goal and its context. It also unfolds the Caduceus model of floating-point numbers, with its floating-point, exact and model parts.

Once all the Caduceus notations have been turned into Gappa's ones, the Gappa tool can be called to finish the proof. In order to do so, the `gappa` tactic first puts the goal into a form suitable to Gappa. This transformation is performed by the `gappa_prepare` sub-tactic written in Ltac, the tactic language embedded into Coq. For instance, if an hypothesis states that $|e| \leq 3/8$, it is replaced by $0 \cdot 2^0 \leq |e| \leq 3 \cdot 2^{-3}$. Indeed, Gappa expects expressions to be bounded on both sides by binary floating-point numbers. The tactic also transforms sub-expressions that are not directly understood by Gappa. For instance, the tactic replaces the multiplicative inverse x^{-1} of Coq by the quotient $1/x$ that both Gappa and Coq understand. Transforming constant expressions is also helpful. Indeed, when the user types the real number 11, Coq implicitly stores it as $1+(1+1)\times(1+(1+1)\times(1+1))$. So the tactic computes the actual integer this expression is equal to instead of sending the expanded expression to Gappa. Conversely, it makes sure that the integers Gappa will later generate are understood by Coq.

In order to perform these transformations, the tactic could perform some pattern-matching to find all the sub-terms that look unadapted and apply rewriting theorems to them. This method is easy to implement but slow, as a huge number of rewriting operation may be needed, especially for constants. Instead, the tactic builds an inductive object that represents the syntax tree of the expressions. Some Coq functions (defined in the logic language, not in the tactic language) then implement the transformations above. They have been proved to generate a syntax tree whose evaluation as a real-valued expression gives the same result as the previous expression. Hence applying this single theorem is enough to get a suitable goal. In other words, the tactic simplifies the goal by convertibility and reflexivity [20,13], which is both time- and space-efficient.

While simplifying the goal, the tactic also replaces all the expressions that are outside of Gappa's scope, *e.g.* $\exp(x)$, by fresh free variables. Once the goal is in a suitable form, the Ocaml part of the tactic takes over. It simply visits the nodes of the simplified syntax tree and produces the corresponding Gappa script.

The script is then sent to Gappa, which either fails or produces a Coq script proving the property. In the latter case, Coq reads this generated script, produces the corresponding λ-term,[12] and matches its type to the simplified goal, hence proving the goal. This means that the `gappa` tactic, while calling an external prover, does produce a complete Coq proof of the goal.

[12] Actually, Coq is unable to deal with two scripts at once. So the tactic first launches a separate Coq session that produces a λ-term in the context of Gappa's libraries. Then it runs another separate session to get a λ-term with fully-qualified names and no notations. This last λ-term can finally be loaded in the original user session, without interfering with user-defined names and notations. This inefficiency could be fixed in two ways: enhance Coq so that separate sessions are no longer needed for reading scripts, or enhance Gappa so that it directly produces a plain λ-term.

5 Conclusion

We have presented an integration of the Gappa automated prover in the Coq proof assistant. This greatly eases the verification process of numerical programs. As shown with realistic examples, the `gappa` tactic significantly reduces the size of Coq proofs, and improves their maintainability. This tactic is available from the V8.2 Coq standard distribution.

This paper focused on C programs verified with the Caduceus tool. However, this approach is generic enough to apply to other verification tools, such as Frama-C[13] for C programs or Krakatoa for Java programs [2].

The current `gappa` tactic does not encompass all the features of the Gappa tool. In particular, there is no way to pass hints to Gappa, such as interval bisection or equalities. Another unavailable feature is the ability of Gappa to infer ranges for particular expressions. This could be used to relieve the user from the burden of guessing logical cuts.

The Gappa tool is limited to a small logical fragment dedicated to floating-point arithmetic, and so is the `gappa` tactic. A more ambitious perspective is to integrate Gappa to a state-of-the-art SMT solver such as Alt-Ergo [4]. This would result in more VCs discharged automatically but also in more automation when invoked from Coq.

References

1. Microprocessor Standards Subcommittee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (August 2008) 1–58
2. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: 19th International Conference on Computer Aided Verification. Volume 4590 of Lecture Notes in Computer Science., Berlin, Germany, Springer (July 2007) 173–177
3. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In Davies, J., Schulte, W., Barnett, M., eds.: 6th International Conference on Formal Engineering Methods. Volume 3308 of Lecture Notes in Computer Science., Seattle, WA, USA, Springer (November 2004) 15–29
4. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantical Combination of Congruence Closure with Solvable Theories. In: Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007). Volume 198-2 of Electronic Notes in Computer Science., Elsevier Science Publishers (2008) 51–69
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580,583
6. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)

[13] Available at `http://frama-c.cea.fr/`.

8. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.2. (2008) `http://coq.inria.fr`.
9. Mayero, M.: The Three Gap Theorem (steinhauss conjecture). In: Proceedings of TYPES'99. Volume 1956., Springer-Verlag LNCS (2000) 162–173
10. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland (2001) 169–184
11. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (November 2004)
12. Boldo, S.: Pitfalls of a Full Floating-Point Proof: Example on the Formal Proof of the Veltkamp/Dekker Algorithms. In: Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR), Seattle, USA (August 2006) 52–66
13. Melquiond, G.: Proving bounds on real-valued functions with computations. In Armando, A., Baumgartner, P., Dowek, G., eds.: Proceedings of the 4th International Joint Conference on Automated Reasoning. Volume 5195 of Lecture Notes in Artificial Intelligence., Sydney, Australia (2008) 2–17
14. de Dinechin, F., Lauter, C., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France (2006) 1318–1322
15. Melquiond, G.: De l'arithmétique d'intervalles à la certification de programmes. PhD thesis, École Normale Supérieure de Lyon, Lyon, France (2006)
16. Bécache, E.: Étude de schémas numériques pour la résolution de l'équation des ondes. ENSTA (September 2003)
17. Kahan, W.: On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic. World-Wide Web document (November 2004)
18. Boldo, S., Daumas, M., Kahan, W., Melquiond, G.: Proof and certification for an accurate discriminant. In: 12th IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, Duisburg, Germany (September 2006)
19. Boldo, S.: Kahan's algorithm for a correct discriminant computation at last formally proven. IEEE Transactions on Computers **58**(2) (February 2009) 220–225
20. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software. (1997) 515–529