Making Proofs of Floating-Point Programs Accessible to Regular Developers

Claire Dross and Johannes Kanig

AdaCore, F-75009 Paris

Abstract. Formal verification of floating-point computations remains a challenge for the software engineer. Automated, specialized tools can handle floating-point computations well, but struggle with other types of data or memory reasoning. Specialized tools based on proof assistants are very powerful, but require a high degree of expertise. General-purpose tools for deductive verification of programs have added support for floating-point computations in recent years, but often the proved properties are limited to verifying ranges or absence of special values such as NaN or Infinity. Proofs of more complex properties, such as bounds on rounding errors, are generally reserved to experts and often require the use of either a proof assistant or a specialized solver as a backend.

In this article, we show how generic deductive verification tools based on general-purpose SMT solvers can be used to verify different kinds of properties on floating point computations up to bounds on rounding errors. The demonstration is done on a computation of a weighted average using floating-point numbers, using an approach which is heavily based on auto-active verification. We use the general-purpose tool SPARK for formal verification of Ada programs based on the SMT solvers Alt-Ergo, CVC4, and Z3 but it can in principle be carried out in other similar languages and tools such as Frama-C or KeY.

Keywords: Deductive verification · Numerical computation · Numerical precision.

1 Introduction

Floating-point computations are at the heart of many critical systems. However, reasoning on floating-point numbers is generally agreed to be counterintuitive. For example, most usual properties on arithmetic operations, such as associativity, no longer hold because of rounding errors [21]. Therefore, floating-point reasoning is an area where tool-assisted verification is welcome.

Deductive verification of programs has improved significantly over the years, and many popular programming languages have their own tool (Java [2,10,17], C [12,17], Rust [3,19], Ada [20]...). Still, floating-point computations remain a challenge in this area. On the one hand, formal verification tools specialized in floating-point computations and error bounds [14,13] generally can only verify

the properties they were designed for and cannot take into account separate considerations, such as data-structures, or pointer manipulation. On the other hand, even if some general purpose verification tools support floating-point arithmetic (Frama-C [12], SPARK [16], KeY [1]), they offer less automation in this domain, and the verified properties generally remain limited to ranges and absence of special values, though more precise properties have been considered in the context of the combination of a deductive verification tool with an automated solver specialized in floating-point reasoning (for example using the solver Gappa inside Frama-C [7,6]).

In this article, we want to demonstrate that properties such as error bounds are within reach for deductive verification tools in what we believe is the most common set-up: a generic deductive verification tool with general-purpose SMT solvers as a backend. We use the SPARK tool [20] for the formal verification of a subset of the Ada language [4] with the SMT solvers Alt-Ergo, CVC4, and Z3 as a backend, but the approach could in principle be carried out in other systems.

It is expected that a certain amount of user interaction will be necessary for the verification, especially as we go further in the complexity of the properties we are trying to verify. We choose to stick to auto-active verification only, as opposed to, for example, going to a proof assistant, so that we remain in the range of what a software developer might be expected to do. However we take advantage of the lemma library of SPARK containing lemmas which are themselves verified using the Coq proof assistant. The lemma library is publicly available with the source code of SPARK [1].

After a quick presentation of SPARK, this paper focuses on explaining how to verify increasingly complex properties on floating-point computations while remaining in a classical developer set-up. In Section 3, we stick to the verification of absence of special values and bounds on program ouputs, like is done in most industrial uses of deductive verification tools. Section 4 goes further and considers bounds on the rounding error of a computation based only on basic arithmetic operations. This work takes advantage of the recently added Ada library for infinite precision arithmetic operations on rational numbers. In Section 5, we consider extending this reasoning to operations which do not benefit from a precise support in the verification tool (and/or return irrational values). We take square root as an example.

2 SPARK Ada

SPARK is an environment for the verification of Ada programs used in critical software development. The SPARK language subset and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems. The SPARK toolset can prove the absence of run-time errors such as arithmetic and buffer overflow, division by zero, and pointer manipulation. SPARK verification is modular at the sub-

[1] https://github.com/AdaCore/spark2014/tree/master/include

program[2] level, so each subprogram is verified independently from its calling context. As a consequence, annotations on subprograms are usually required. In SPARK, this is achieved with contracts - mainly pre- and postconditions. The SPARK tool checks the correctness of the user-provided contracts as well as the absence of run-time errors.

To avoid having to learn a separate language for annotations, all annotations in SPARK, including assertions and pre- and postconditions, use the syntax and semantics of regular Ada boolean expressions. As a consequence, they are executable and can be checked at runtime using a compiler switch. However, the annotation language is somewhat restricted compared to a non-executable logic language. For example, one cannot quantify over arbitrary data, only ranges, enumerations or contents of containers.

Under the hood, SPARK translates the Ada program into a WhyML program suitable for processing by the Why3 tool [5]. Why3 then generates logical formulas, called proof obligations, whose validity implies the correctness of the program. Automatic theorem provers are then used to discharge these proof obligations. SPARK ships with three built-in SMT solvers CVC4, Z3, and Alt-Ergo. For floating-point variables and basic arithmetic operations, SPARK assumes conformance to the IEEE 754 floating-point standard, which allows the tool to take advantage of the recent built-in floating-point support in SMT solvers [23].

The running example used throughout the article is presented in Figure 1. It is a function doing the weighted average of an array of floating-point numbers. It uses single precision floating-point values (type `Float` in SPARK), but similar results have been obtained with double precision floats without more proof efforts. The example uses recursive functions to make it shorter. However the implementation can easily be rewritten as a loop, while keeping the current definition as a specification.

3 Proof of integrity of floating-point operations

3.1 Floating point support in SPARK

SPARK supports basic floating-point operations natively and assumes IEEE 754 semantics for them. However, the special value NaN, as well as positive and negative infinity, are excluded. Generating such a value in a floating-point computation is considered to be an error, and will result in an unproved check. In practice, this amounts to checking that we never divide by zero and that computations do not overflow. Because of these semantics, SPARK reports failed overflow checks on our running example, one on each arithmetic operation.

One possible way to deal with these potential overflows is to replace problematic operations by saturating operations, which never overflow but yield potentially incorrect results. This is an easy solution that may be appropriate for

[2] Ada uses the term *subprogram* to denote functions (which return a value) and procedures (which do not).

```
package Libst with SPARK_Mode is
   Max_Index : constant := 100;
   --  Number of elements in an array of values
   subtype Extended_Index is Natural range 0 .. Max_Index;
   subtype Index is Positive range 1 .. Max_Index;

   type Value_Array is array (Index) of Float;
   type Weight_Array is array (Index) of Float;

   --  Definition of the weighted average on an array of values using floating
   --  point computation.

   function Sum_Weight_Rec
     (Weights : Weight_Array; I : Extended_Index) return Float
   is (if I = 0 then 0.0 else Sum_Weight_Rec (Weights, I - 1) + Weights (I));
   function Sum_Weight (Weights : Weight_Array) return Float is
     (Sum_Weight_Rec (Weights, Max_Index));
   --  Sum of an array of weights

   function Weighted_Sum_Rec
     (Weights : Weight_Array;
      Values  : Value_Array;
      I       : Extended_Index) return Float
   is (if I = 0 then 0.0
       else Weighted_Sum_Rec (Weights, Values, I - 1) + Weights (I) * Values (I));
   function Weighted_Average
     (Weights : Weight_Array; Values : Value_Array) return Float
   is
     (Weighted_Sum_Rec (Weights, Values, Max_Index) / Sum_Weight (Weights))
   with Pre ⇒ Sum_Weight (Weights) ≠ 0.0;
   --  Weighted average of an array of values
end Libst;
```

Fig. 1. SPARK function computing the weighted average of values in an array

certain use cases, but it changes the algorithm if the saturation happens in practice. In this paper, we instead select bounds for the inputs and intermediate variables such that overflows never occur, and attempt to formally verify their absence.

3.2 Bounds for floating-point types

In general, large negative and positive floating-point inputs need to be excluded to avoid overflows. In the presence of division, it might also be necessary to exclude values that are too close to zero, as the division might overflow in that case. The bounds for the input values can come from the application context (for example, a variable representing an angle can be limited to the range between 0 and 360°), or might be dictated by the needs of the verification of the algorithms.

Ada (and SPARK), in addition to the built-in `Float` and `Long_Float` (double precision) types, supports the definition of user-defined subtypes. These subtypes can specify a more restrictive range using the `range` keyword, or restrict the set of allowed values using a `Predicate` (or both). It is generally good practice to document application-specific ranges using subtypes when possible, instead of using the built-in types everywhere.

In our example, we choose to bound values so as to specify our algorithm in a more elegant way. We want to express the maximal weighted sum using a product `Max_Value` × `Max_Index`. For this multiplication to be equal to a sequence of additions, all the intermediate values should be in the range in which each integer value is representable exactly as a floating-point value. This range is bounded by 2^{24}, so the largest value we allow is $2^{24}/$`Max_Index`.

We arbitrarily choose to restrict weights to the range between 0 and 1. It is a common enough use-case for applications of the weighted average. However, excluding zero from the sum of the weights for the division is not enough to guarantee the program integrity - if we divide by a very small value, we might get an overflow. That's why we also exclude very small non-zero values from the range of the weights. As a reasonable bound[3], we use 2^{-63}. We still allow individual weights to be zero. The range of weights is thus non-contiguous and is expressed through a predicate stating that a weight is either zero, or is in the range from `Min_Weight` to 1.

```
Max_Exact_Integer_Computation : constant := 2.0 ** 24;
--  Upper bound of the interval on which integer computations are exact
Float_Sqrt : constant Float := 2.0 ** 63;
--  Safe bound for multiplication

Max_Value : constant :=
  Float'Floor (Max_Exact_Integer_Computation / Float (Max_Index));
--  Biggest integer value for which the sum is guaranteed to be exact
subtype Value is Float range -Max_Value .. Max_Value;
type Value_Array is array (Index) of Value;

Min_Weight : constant := 1.0 / Float_Sqrt;
--  Avoid values too close to zero to prevent overflow on divisions
subtype Weight is Float range 0.0 .. 1.0 with
  Predicate ⇒ Weight in 0.0 | Min_Weight .. 1.0;
type Weight_Array is array (Index) of Weight;
```

3.3 Proving the absence of overflows

As deductive verification is modular on a per-subprogram basis, it is necessary to annotate all our subprograms with bounds for their floating-point inputs and outputs. This can be done in two different ways. It is possible to create new bounded subtypes like the ones we introduced for values and weights for intermediate values in the computation. However, as SPARK does not support dependent types, this only works if the bounds do not depend on other inputs / outputs of the subprogram. Another alternative is to specify the bounds in the contract (pre- and postcondition) of the subprogram.

In our example, we define a new bounded type for the result of `Sum_Weight`. Since weights are always less than 1, the sum of the weights is less than the number of elements in the array. As we need to divide by this sum in the computation of our weighted average, we also need to provide a lower bound for this sum when it is not zero. As for the weights, we can say that the sum is either 0 (if all the

[3] It is the bound used for lemmas about division in the lemma library of SPARK (more about the lemma library later in this article)

weights are 0) or at least Min_Weight (it would be exactly Min_Weight if a single weight has value Min_Weight and all others have value 0).

```
subtype Sum_Of_Weights is Float range 0.0 .. Float (Max_Index) with
  Predicate ⇒ Sum_Of_Weights in 0.0 | Min_Weight .. Float (Max_Index);

function Sum_Weight (Weights : Weight_Array) return Sum_Of_Weights is
  (Sum_Weight_Rec (Weights, Max_Index));
--  Sum of an array of weights
```

Given the recursive definition of Sum_Weight_Rec, using Sum_Of_Weights as a subtype for its result is not enough, as assuming the predicate for the return value of the recursive call would not be sufficient to prove the predicate for the result of the addition. Instead, we add a postcondition to Sum_Weight_Rec which bounds the computation depending on the current index.

```
function Sum_Weight_Rec
  (Weights : Weight_Array; I : Extended_Index) return Float
is (if I = 0 then 0.0 else Sum_Weight_Rec (Weights, I - 1) + Weights (I))
with Post ⇒ Sum_Weight_Rec'Result in 0.0 | Min_Weight .. Float (I);
```

The functions Weighted_Average and Weighted_Sum_Rec can be annotated in a similar way. The function Weighted_Sum_Rec is constrained via a postcondition while Weighted_Average has a constrained return type:

```
 function Weighted_Sum_Rec
  (Weights : Weight_Array;
   Values  : Value_Array;
   I       : Extended_Index) return Float
is  (if I = 0 then 0.0
     else Weighted_Sum_Rec (Weights, Values, I - 1) + Weights (I) * Values (I))
with
  Post ⇒ Weighted_Sum_Rec'Result in
    -(Max_Value * Float (I)) .. Max_Value * Float (I);

Max_Sum_Of_Values : constant := Max_Value * Float (Max_Index) / Min_Weight;
subtype Sum_Of_Values is Float range -Max_Sum_Of_Values .. Max_Sum_Of_Values;
function Weighted_Average
  (Weights : Weight_Array; Values : Value_Array) return Sum_Of_Values
is
  (Weighted_Sum_Rec (Weights, Values, Max_Index) / Sum_Weight (Weights))
with Pre ⇒ Sum_Weight (Weights) ≠ 0.0;
--  Weighted average of an array of values
```

Note that the bounds used in the Sum_Of_Values subtype for the result of Weighted_Average are far from the bounds of the mathematical computation (which is known to stay between −Max_Value and Max_Value). Better bounds can be obtained by distributing the division over the addition to compute $w_1/\texttt{Sum\_Weight} * v_1 + \ldots$. The fact that each weight is smaller than their sum could then be used to reduce Max_Sum_Of_Values to Max_Value * Float(Max_Index). However, as we don't need this improved bound to prove the absence of overflows here, we have decided against changing the formulation.

With these annotations, the tool should theoretically have enough information to verify our program. As it is, the postconditions of Sum_Weight_Rec and Weighted_Sum_Rec remain unproved. This is because reasoning about floating-point computation is a well known challenge, both for programmers and for verification tools. If we want SPARK to automatically verify our subprograms, we will have to help it, using auto-active verification through ghost code.

Bounding floating-point operations: Leino and Moskal coined the term of auto-active verification [18] as a middle ground between fully automatic verification with no user input, and fully interactive verification as with a proof assistant. The idea is to add extra annotations and code to a program purely for the sake of verification. Auto-active verification can come in many forms. The most basic example is adding assertions that can help "cut" a difficult proof into two easier proofs. SPARK supports the notion of ghost code, where variables and subprograms can be marked as ghost [15]. The compiler checks that such code cannot influence the functional behavior of non-ghost code, and strips such code from the program during compilation when runtime assertion checking is disabled.

In SPARK, there is no built-in way to name or parameterize assertions to create reusable lemmas or axioms. An alternative way to share assertions is to use ghost procedures. The premises of the lemma we want to parameterize should be put as preconditions, whereas its conclusion should be used as a postcondition:

```
procedure Lemma (Param1 : T1; Param2 : T2, ...)
with Pre  ⇒ Premise1 and then Premise2 and then ...,
     Post ⇒ Conclusion,
     Global ⇒ null,
     Ghost;
```

As SPARK handles subprograms modularly, it will check that the postcondition of `Lemma` follows from its precondition in any context during its verification. When the procedure is called, it will check the precondition - the premises, and assume the postcondition - the conclusion. As a convention, such *lemma procedures* or simply *lemmas* have the prefix `Lemma_` in their name, are marked as ghost, and have no global effects.

The body (implementation) of a lemma procedure may contain any code or ghost code that helps to establish the postcondition, such as calls to other lemma procedures, or even loops to establish an inductive proof. In simple cases the body of the lemma procedure may be empty, because the provers can prove the postcondition from the precondition directly. Note that it is often the case that provers can prove a lemma's postcondition directly, but would not be able to prove an equivalent assertion in the middle of a large subprogram, due to the presence of a large, mostly irrelevant context.

SPARK comes with a *lemma library* that among other things offers basic lemmas to deduce bounds on the result of floating-point operations. Some of these lemmas are proved automatically by the SMT solvers, like the lemma `Lemma_Add_Is_Monotonic` below[4]. Others are proved using a proof assistant, like the similar `Lemma_Mult_Is_Monotonic` for multiplication.

```
procedure Lemma_Add_Is_Monotonic
  (Val1 : Float;
   Val2 : Float;
   Val3 : Float)
with
  Global ⇒ null,
  Pre  ⇒
```

[4] Lemma `Lemma_Add_Is_Monotonic` takes inputs between `Float'First / 2.0` and `Float'Last / 2.0` to statically ensure absence of overflows in the addition.

```
    (Val1 in Float'First / 2.0 .. Float'Last / 2.0) and then
    (Val2 in Float'First / 2.0 .. Float'Last / 2.0) and then
    (Val3 in Float'First / 2.0 .. Float'Last / 2.0) and then
     Val1 ≤ Val2,
  Post ⇒ Val1 + Val3 ≤ Val2 + Val3;
```

Coming back to our example, we use a call to `Lemma_Add_Is_Monotonic` in the proof of `Sum_Weight` to bound the result of the addition by the sum of the bounds:

```
Lemma_Add_Is_Monotonic
    (Sum_Weight_Rec (Weights, I - 1), Float (I - 1), Weights (I));
```

Precise bounds on integers: To prove the integrity of floating-point computations, it might be necessary to know that the computation is exact in special cases. In our examples, the bounds we have deduced earlier on our floating-point computations are not enough to prove the postconditions as they are stated. Indeed, in the postconditions, we have chosen to use a multiplication instead of a sequence of additions, which is not equivalent in general with floating-point numbers because of rounding errors. However, the floating-point computations are sometimes known to be exact, in particular for computations on integers if the values are small enough to fall in a range where all integer values are representable. It is the case in our example, both for the summation from 1 to `Max_Index` in `Sum_Weight` and for the additions on `Max_Value` in the postcondition of `Weighted_Average` (`Max_Value` has been chosen for that).

To help SPARK deduce that the computations are exact in our example, we introduce specific lemmas. The lemma `Lemma_Add_Exact_On_Index` states that floating-point addition is exact on integers in the range of the index type.

```
procedure Lemma_Add_Exact_On_Index (I, J : Natural) with
  Ghost,
  Pre  ⇒ I ≤ Max_Index and J ≤ Max_Index,
  Post ⇒ Float (I) + Float (J) = Float (I + J);
--  Floating-point addition is exact on indexes
```

These lemmas bring the remaining missing pieces for the proof of our example. However, to be able to call them in the bodies of our recursive functions, we need to turn them into regular functions with a proper body. The `Subprogram_Variant` annotation is unrelated. It is used to prove termination of recursive functions.

```
function Sum_Weight_Rec
  (Weights : Weight_Array;
   I       : Extended_Index) return Float
with
  Subprogram_Variant ⇒ (Decreases ⇒ I),
  Post ⇒ Sum_Weight_Rec'Result =
    (if I = 0 then 0.0 else Sum_Weight_Rec (Weights, I - 1) + Weights (I))
    and then Sum_Weight_Rec'Result in 0.0 |  Min_Weight .. Float (I);

function Sum_Weight_Rec
  (Weights : Weight_Array;
   I       : Extended_Index) return Float is
begin
  if I = 0 then
    return 0.0;
```

```
  else
    Lemma_Add_Is_Monotonic
      (Sum_Weight_Rec (Weights, I - 1), Float (I - 1), Weights (I));
    Lemma_Add_Exact_On_Index (I - 1, 1);
    return Sum_Weight_Rec (Weights, I - 1) + Weights (I);
  end if;
end Sum_Weight_Rec;
```

4 Functional correctness

To go further in the verification process, functional contracts should be added to specify the expected behavior of the program. In general, functional contracts can take various forms. On simple programs doing only floating-point computations, it is interesting to compare the floating-point result with the result using exact computations on real numbers. Aside from the direct interest of having an upper bound on the rounding error in the computation, such a specification allows us to lift properties that are known to hold for the real computation. As an example, the bounds given for `Weighted_Average` in the previous section are obviously greatly over-approximated. On real numbers, the weighted average of an array of values is known to stay in the bounds of the provided values. This is not true for the floating-point computation, even with a well behaved bound like `Max_Value`, because of the rounding errors. Bounding the rounding error in the computation of the weighted average would allow us to tighten these bounds efficiently.

4.1 The `Big_Real` library

To write specifications describing rounding errors, contracts need to use operations with unlimited precision. As assertions in SPARK are executable, it is not possible to directly use an axiomatic definition of real numbers as can be done for example for Why3 [5] or Frama-C [12]. However, in the upcoming release of Ada (scheduled for 2022) new libraries for infinite precision integers and rational numbers have been added[5]. To specify our example, rational numbers are enough. In the next section, we discuss how the approach can be extended to irrational computations.

The Ada library for infinite precision rational numbers defines a type named (rather counterintuitively) `Big_Real` as the quotient of two infinite precision integers. It provides the usual operations on rational numbers, as well as conversion functions from floating-point and integer types. This library benefits from built-in support in the SPARK tool. In the generation of proof obligations, objects of type `Big_Real` are translated as mathematical real numbers and their operations are the corresponding real operations. This allows SPARK to benefit from the native support from the underlying solvers.

[5] http://ada-auth.org/standards/2xrm/html/RM-A-5-7.html

4.2 Specifying the rounding error of floating-point computations

Contracts are expressed in terms of specification functions doing computations on rational numbers. These functions are in a package marked as `Ghost`, so that they will be eliminated if the code is compiled with assertions disabled. It avoids linking the `Big_Reals` library into production code.

```
function R_Weighted_Sum_Rec
  (Weights : Weight_Array;
   Values  : Value_Array;
   I       : Extended_Index) return Big_Real
is
  (if I = 0 then 0.0
   else R_Weighted_Sum_Rec (Weights, Values, I - 1)
     + To_Big_Real (Weights (I)) * To_Big_Real (Values (I)))
with Subprogram_Variant ⇒ (Decreases ⇒ I);

function R_Weighted_Average
  (Weights : Weight_Array; Values : Value_Array) return Big_Real
is
  (R_Weighted_Sum_Rec (Weights, Values, Max_Index) / R_Sum_Weight (Weights))
with Pre ⇒ R_Sum_Weight (Weights) ≠ 0.0;
--  Weighted average of an array of values
```

The rounding error performed during the computation is expressed as the difference between the floating-point function and the infinite precision one. The postcondition on the floating-point function could theoretically be extended with an estimation of the rounding error. In our example, we prefer to use separate lemmas. The procedure `Error_For_Average` below gives an upper bound on the rounding error in the computation of the weighted average. It is called explicitly whenever we need to approximate this bound in our verification.

```
procedure Lemma_Error_For_Average
  (Weights : Weight_Array; Values : Value_Array)
with
  Pre  ⇒ R_Sum_Weight (Weights) ≠ 0.0,
  Post ⇒ abs (To_Big_Real (Weighted_Average (Weights, Values)) -
                R_Weighted_Average (Weights, Values)) ≤ ???;
--  Error bound on the computation of Weighted_Average
```

Evaluating the error bound: The first step in the specification process is to determine a bound for the rounding error in the computation. There are tools that can be used to predict it [13,14]. In SPARK, there is no such facility, so the bound needs to be computed manually from the individual operations done in the code. Note that this manual effort of deconstruction is also useful in the next step to help the verification tool. Here this is done through ghost code, but a similar reasoning could be expected if a proof assistant were used.

The basic steps to estimate the error bound are rather automatic. The IEEE 754 standard specifies that rounding on binary operations is always done to the closest floating-point number. As a result, the rounding error is less than half of the difference between two consecutive floating-point values. As the difference between two consecutive floating-point values varies depending on the value of the floating-point, the rounding error is generally bounded relative to the result of the operation. Standard floating-point numbers are spaced logarithmically,

so this bound is a constant named `Epsilon` multiplied by the result of the operation. On 32-bit floating-point numbers, `Epsilon` is 2^{-24}, that is, roughly 10^{-7}. Subnormal numbers are linearly spaced, so the error on them is at most half of the smallest subnormal number [21].

In our example, the relative error for the function `Sum_Weight` is bounded by `Max_Index` × `Epsilon`. Indeed, the rounding error for each addition but the first (with 0) is proportional to the partial sum at the point of the addition, which is less than the complete sum[6].

```
procedure Lemma_Error_For_Sum_Weight (Weights : Weight_Array)
with Post ⇒
   abs (To_Big_Real (Sum_Weight (Weights)) - R_Sum_Weight (Weights))
   ≤ 1.0E-5 * R_Sum_Weight (Weights);
--  Error bound on the computation of Sum_Weight
```

In the computation of `Weighted_Average`, the values can be both negative and positive, so we cannot exclude subnormal numbers[7]. Thus, the error bound will have two parts, a relative part for standard numbers, and an absolute one for subnormals. The relative error bound is expressed in terms of the weighted average of the absolute value of the values.

```
procedure Lemma_Error_For_Average
  (Weights : Weight_Array; Values : Value_Array)
with
  Pre  ⇒ R_Sum_Weight (Weights) ≠ 0.0,
  Post ⇒ abs (To_Big_Real (Weighted_Average (Weights, Values)) -
                R_Weighted_Average (Weights, Values))
    ≤ 1.01E-45 + 2.03E-43 / R_Sum_Weight (Weights)
           + 2.05E-5 * R_Weighted_Average_Abs (Weights, Values);
--  Error bound on the computation of Weighted_Average
```

The error bound is composed of three parts. The first one comes from the absolute error on the division, the second is the absolute error on the computation of the sum of values (100 additions and 100 multiplications), and the third is the sum of the relative errors on the sum of weights and the sum of values (roughly 1.0E-5 for the sum of weights, see above, and the same for the values, plus a bit more for the division itself).

Lifting properties from the real computation: Aside from the direct interest in having an upper bound on the rounding error in the computation, this specification allows us to lift properties that are known to hold for the real computation. For example, it is well-known that the weighted average of an array of real values ranging from -`Max_Value` to `Max_Value` is also between -`Max_Value` and `Max_Value`. Using this fact, we might want to compute a better bound for

[6] Note that we use 10^{-7} as an approximation of `Epsilon` here. A more precise approximation could be considered, like 6×10^{-8}, but, from our experiments, it causes proofs of error bounds on basic operations to become out of reach of the automated SMT solvers used behind SPARK.

[7] Theoretically, as addition on standard numbers can only give standard numbers and addition on subnormals is exact, the absolute part of the bound should not be necessary here. However, this reasoning is currently out of reach of the underlying SMT solvers.

the result of `Weighted_Average` than the current one (`Max_Sum_Of_Values` is larger than 1.5e26 against 1.7e5 for `Max_Value`). The expected property is proved on the real computation, and then lifted to the floating-point computation using the approximation of error bounds.

The error bound provided by the lemma `Lemma_Error_For_Average` is maximal when the denominator is close to `Min_Weight` and the weighted average is close to `Max_Value`. In this case, the significant part is the relative error on the weighted average. It evaluates to less than 3.5 as an absolute value. This allows us to prove the following lemma, providing notably more precise bounds for the computation.

```
procedure Lemma_Precise_Bounds_For_Average
  (Weights : Weight_Array; Values : Value_Array)
with
  Pre  ⇒ Sum_Weight (Weights) ≠ 0.0,
  Post ⇒ Weighted_Average (Weights, Values)
     in - (Max_Value + 3.5) .. Max_Value + 3.5;
--  Precise bounds for Weighted_Average obtained through error bound computation
```

Note that this bound is not optimal, as we have approximated the error bounds in the computations. In particular, we are losing nearly a factor two on the approximation of `Epsilon`. However, some testing shows that the optimal bound is at least `Max_Value` + 1.3, so it is not too far off either.

Proving that the error bounds are correct: Correctly predicting the error bounds is not enough for SPARK to verify them. As in Section 3, reasoning about floating-point computations is difficult for the underlying solvers, and it is even worse when considering a combination of floating-point and real numbers. As before, it is possible to help the tool using auto-active verification. The key is to work in small steps, factoring out each proof step in a lemma.

As a basis, shared lemmas are introduced to bound the rounding error performed by a single floating-point operation. If the result is a subnormal number, the error is less than half of the smallest positive subnormal number. If it is a standard number, it can be bounded linearly using the `Epsilon` constant.

```
procedure Lemma_Bound_Error_Add (X, Y : Floats_For_Add) with
  Post ⇒ abs (To_Big_Real (X + Y) - (To_Big_Real (X) + To_Big_Real (Y))) ≤
    (if abs (To_Big_Real (X) + To_Big_Real (Y)) ≥ First_Norm
     then Epsilon * abs (To_Big_Real (X) + To_Big_Real (Y))
     else Error_Denorm);
```

Complex computations are split into parts, each part accounting for an individual error factor. For example, the computation of `Weighted_Average` is made of three parts: the computation of the weighted sum in the numerator, the computation of the sum of the weights in the denominator, and the division itself. For the tool to be able to put the parts together, the global error needs to be expressible as a sum of the various parts. The following code gives a possible partition of the computation of `Weighted_Average`:

```
Num_F : constant Float := Weighted_Sum_Rec (Weights, Values, Max_Index);
Den_F : constant Float := Sum_Weight (Weights);
Num_R : constant Big_Real := R_Weighted_Sum_Rec (Weights, Values, Max_Index);
Den_R : constant Big_Real := R_Sum_Weight (Weights);
```

```
abs (To_Big_Real (Num_F / Den_F) - Num_R / Den_R) ≤
    --  Error on the division
    abs (To_Big_Real (Num_F/Den_F) - (To_Big_Real (Num_F)/To_Big_Real (Den_F)))
    --  Error on the computation of the numerator
    + abs (To_Big_Real (Num_F)/To_Big_Real (Den_F) - Num_R/To_Big_Real (Den_F))
    --  Error on the computation of the denominator
    + abs (Num_R / To_Big_Real (Den_F) - Num_R / Den_R)
```

As can be seen in the example, the parts are not necessarily direct applications of one of the basic lemmas, or one of the lemmas introduced for previous computations. For the second term of the addition, the lemma introduced for `Weighted_Sum_Rec` will bound the difference between `Num_F` and `Num_R`. The effect of dividing each term by `To_Big_Real (Den_F)` must still be accounted for separately. New lemmas can be introduced for that. The premises are the simple error we are trying to compute, and the errors on other terms mentioned in the expression. The conclusion is the error on the considered operation. It should be expressed in terms of the real computation only.

```
--  Lemma to compute the part of the error in Weighted_Average coming from the
--  computation of the numerator.
procedure Lemma_EB_For_Sum
  (Num_F               : Floats_For_Mul;
   Den_F               : Floats_For_Div;
   Num_R, Den_R, Num_A : Big_Real)
with
  Pre  ⇒ Den_F > 0.0 and Den_R > 0.0 and Num_A ≥ abs (Num_R)
    --  Error on the computation of the numerator
    and abs (To_Big_Real (Num_F) - Num_R) ≤ 2.01E-43 + 1.01E-5 * Num_A
    --  Error on the computation of the denominator
    and abs (To_Big_Real (Den_F) - Den_R) ≤ 1.0E-5 * Den_R,

  --  Error accounting for the error on the computation of the numerator
  Post ⇒ abs (To_Big_Real (Num_F)/To_Big_Real (Den_F) - Num_R/To_Big_Real (Den_F))
    ≤ 2.02E-43 / Den_R + 1.02E-5 * Num_A / Den_R;
```

The statement and auto-active proof of the lemmas providing the error bounds on the computations involved in the evalutation of the weighted average amount to a bit less than 400 lines of ghost code, while the initial (unproved) implementation was 25 lines of code and its real counterpart is 34 lines of code.

4.3 Prover performance

Encoding The SPARK tool translates Ada code to Why3, which then generates proof obligations for various provers. The three provers CVC4, Z3 and Alt-Ergo are available in SPARK by default and have been used in our work. CVC4 and Z3 use the SMT-LIB encoding of Why3. This means that floating-point variables and operations are encoded using the corresponding SMT-LIB operations [16]. For Alt-Ergo, its native input is used, together with an axiomatization based on arithmetic operations on reals, following by rounding [11].

Experimental results Figure 2 contains the experimental results. All three provers are necessary to fully prove the example. If one of the provers was not available, some additional effort, with additional lemmas and assertions, and possibly longer prover timeouts, could also lead to full proofs.

Prover	**Alt-Ergo** 2.3.0	**CVC4** 1.8	**Z3** 4.8.10	Total
Runtime Checks	453 (0)	470 (10)	437(0)	540
Assert, Lemmas, Pre/Post	70 (10)	45 (2)	51(2)	79

Fig. 2. Results of provers on the Weighted Average example on 619 proof obligations. A timeout of 60s was used. A cell contains the number of proved VCs, and the number of VCs only this prover could prove in parentheses. The last column contains the total number of VCs in each category. The results were obtained on an AMD Ryzen 3950x with 64GB of RAM.

One can see that, while CVC4 is strong on runtime checks (mostly array accesses and checks on bounds of integer and floating-point operations on the example), Alt-Ergo is the strongest prover when it comes to lemma procedures, assertions and pre- and postconditions. Alt-Ergo also uniquely proves the most VCs in this category. This probably comes from the fact that Alt-Ergo is the only prover whose support for floating-point arithmetic is not based on bitlevel reasoning, but on an axiomatization using real numbers and rounding, which helps on our example, since it involves both real and floating point computations.

5 Going beyond basic arithmetic operations

The method applied in the previous section works on computations involving only basic arithmetic operations. However, it is not enough in general. For example, the computation of the standard error of our average computation would require the use of square root. Our method can be extended to handle other operations, but it necessitates some additional preparatory steps.

5.1 Approximating irrational computations

The first issue is the lack of a real infinite precision counterpart for the considered floating-point operation. Since SPARK only supports infinite precision rational numbers, and not real numbers, if the operation returns an irrational value, then it is not possible to write a specification function to represent the operation with infinite precision. Fortunately, since we are only interested in the error bounds, it is enough to provide a function giving a good enough approximation of the result as a rational value. Here, we define a function that approximates the square root of a rational number with a given precision. The precision is chosen to be negligible with respect to the rounding errors on the floating-point type.

```
Sqrt_Bound : constant := 1.0E-8;
--  Small enough value to be absorbed by the over-approximation on Epsilon
Sq_Bound   : constant := 1.999 * Sqrt_Bound;
--  Approximation of the minimum bound on the square to ensure Sqrt_Bound

function R_Sqrt (X : Big_Real) return Big_Real with
  Pre  => X >= 0.0,
  Post => R_Sqrt'Result >= 0.0
     and then abs (X - R_Sqrt'Result * R_Sqrt'Result)
       <= Sq_Bound * R_Sqrt'Result * R_Sqrt'Result;
--  Approximation of sqrt on real numbers
```

Coming up with a specification for the rational function might be difficult for transcendental functions. A possibility would be to provide only some properties of the function through axioms (that is, lemma procedures without a body, or whose body is not verified). This would not permit us to benefit from native support in the underlying solvers however. This might not be a big deal, as support for transcendental functions is poor in most solvers. This could be alleviated if needed by providing specialized support for most common transcendental functions in the SPARK tool as it is done for the `Big_Reals` library.

5.2 Axiomatizing floating-point operations

The second issue is the absence of support in the SPARK tool for complex operations on floating-point values. The usual functions on floating-point types (square root, exponentiation, logarithm, trigonometric functions...) are provided as a library. Some of these functions are annotated with a minimal contract providing some bounds on their result, but nothing more precise is known about them by the tool. For example, here is the contract provided for `Sqrt`.

```
function Sqrt (X : Float) return Float with
  Pre  ⇒ X ≥ 0.0,
  Post ⇒ Sqrt'Result ≥ 0.0
    and then (if X = 0.0 then Sqrt'Result = 0.0)
    and then (if X = 1.0 then Sqrt'Result = 1.0)
    and then (if X ≥ Float'Succ (0.0) then Sqrt'Result > 0.0);
```

The reason for this minimal support is that the language does not necessarily enforce compliance to the IEEE 754 standard for these functions on all platforms. When additional information is needed, it can be supplied through axioms. On a safety critical project, particular care should be taken that these axioms are indeed valid on the chosen architecture. Here are some examples of axioms that could be provided for the `Sqrt` function on 32-bit integers:

```
procedure Axiom_Bound_Error_Sqrt (X : Float) with
  Pre  ⇒ X ≥ 0.0,
  Post ⇒ abs (To_Big_Real (Sqrt (X)) - R_Sqrt (To_Big_Real (X))) ≤
    (if Sqrt (To_Big_Real (X)) ≥ First_Norm
     then Epsilon * R_Sqrt (To_Big_Real (X)) else Error_Denorm);

procedure Axiom_Sqrt_Is_Monotonic (X, Y : Float) with
  Pre  ⇒ X ≥ 0.0 and Y ≥ X,
  Post ⇒ Sqrt (Y) ≥ Sqrt (X);

procedure Axiom_Sqrt_Exact_Integer (X : Integer) with
  Pre  ⇒ X in 0 .. 4096,
  Post ⇒ Sqrt (Float (X) ** 2) = Float (X);
```

The first axiom gives an over approximation of the error bound with respect to the approximation of square root on rational numbers. The following two are variants of the lemmas used on basic arithmetic operations to verify program integrity in Section 3. The first states that square root is monotonic while the second expresses that `Sqrt` is exact on the square of a small enough integer. Using these axioms, we have proven correct an implementation of the euclidean

norm of a vector which could be used to compute the standard error of our weighted average[8]. It uses a method similar to the one from Sections 3 and 4.

6 Related Work

Historically, most deductive verification tools interpreted floating-point numbers as real numbers. Following the addition of floating-point support in SMT solvers [11,9], some verification tools changed to a sound handling, even if it meant a decrease in provability. This is the case for Frama-C and SPARK [16], as well as KeY [1] more recently.

Even with built-in support in SMT solvers, verifying floating-point computations remains a challenge. As a result, most verification efforts on floating-point computations rely on either specialized tools or proof assistants to reason about error bounds. For example, the static analyzer Fluctuat [14] is used in an industrial context to automatically estimate the propagation of error bounds in C programs. Specialized libraries are also available inside proof assistants. The Flocq [8] library offers a formalization to reason about floating- and fixed-point computations in Coq. As a middle ground between both approaches, Gappa [13] is a proof assistant targeting specifically verification of floating-point computation, with automated evaluation and propagation of rounding errors.

Verification efforts using standard deductive verification tools largely do not go beyond absence of special values. It is still possible to use a standard verification tool in combination with either a specialized tool or a proof assistant. For example, Gappa was used as a backend for the Frama-C tool to successfully verify an average computation on C code [7,6]. In the opposite direction, a Point-in-Polygon algorithm was verified by first generating lemmas about stable tests using the specialized tool PRECiSA and then reusing them inside Frama-C to prove the program [22].

7 Conclusion

We have demonstrated that it is possible to use general purpose verification tools like SPARK to verify advanced properties of (simple) floating-point programs. Absence of overflows is achievable through bounds propagation. Precise specifications can be written in terms of the equivalent real computation and error bounds, but proving them correct is more involved and requires user interaction currently. The situation could improve in the future, either following improvements in SMT solvers, as support for floating-point in SMT-LIB is fairly recent. Though we believe it stays within the reach of regular developers, our approach does require some expertise, in particular to compute the correct error bounds. It might be possible to use external tools to estimate the bounds and reuse the results, but we have not explored this approach.

[8] https://github.com/AdaCore/spark2014/tree/master/testsuite/gnatprove/tests/U129-014__sqrt_error_bounds

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point Java programs in KeY. In: TACAS (2). pp. 242–261 (2021)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive software verification–the key book. Lecture Notes in Computer Science **10001** (2016)
3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
4. Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64 (2011)
6. Boldo, S.: Formal verification of programs computing the floating-point average. In: International Conference on Formal Engineering Methods. pp. 17–32. Springer (2015)
7. Boldo, S., Marché, C.: Formal verification of numerical programs: from c annotated programs to mechanical proofs. Mathematics in Computer Science **5**(4), 377–393 (2011)
8. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic. pp. 243–252. IEEE (2011)
9. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 79–98. Springer (2019)
10. Cok, D.R.: Openjml: Jml for java 7 by extending openjdk. In: NASA Formal Methods Symposium. pp. 472–479. Springer (2011)
11. Conchon, S., Iguernlala, M., Ji, K., Melquiond, G., Fumex, C.: A three-tier strategy for reasoning about floating-point numbers in smt. In: International Conference on Computer Aided Verification. pp. 419–435. Springer (2017)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c. In: International conference on software engineering and formal methods. pp. 233–247. Springer (2012)
13. De Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using gappa. IEEE Transactions on Computers **60**(2), 242–253 (2010)
14. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 53–69. Springer (2009)
15. Dross, C., Moy, Y.: Auto-active proof of red-black trees in spark. In: NASA Formal Methods Symposium. pp. 68–83. Springer (2017)
16. Fumex, C., Marché, C., Moy, Y.: Automating the verification of floating-point programs. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 102–119. Springer (2017)
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for c and java. In: NASA formal methods symposium. pp. 41–55. Springer (2011)

18. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010)
19. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: Chc-based verification for rust programs. In: European Symposium on Programming. pp. 484–514. Springer, Cham (2020)
20. McCormick, J.W., Chapin, P.C.: Building high integrity applications with SPARK. Cambridge University Press (2015)
21. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems (TOPLAS) **30**(3), 1–41 (2008)
22. Moscato, M.M., Titolo, L., Feliú, M.A., Muñoz, C.A.: Provably correct floating-point implementation of a point-in-polygon algorithm. In: International Symposium on Formal Methods. pp. 21–37. Springer (2019)
23. Rümmer, P., Wahl, T.: An smt-lib theory of binary floating-point arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT). p. 151 (2010)