C-language floating-point proofs layered with VST and Flocq

Andrew W. Appel
Princeton University
and
Yves Bertot
INRIA and Université Côte d'Azur

We demonstrate tools and methods for proofs about the correctness and numerical accuracy of C programs. The tools are *foundational,* in that they are connected to formal semantic specifications of the C operational semantics and of the IEEE 754 floating-point format. The tools are *modular*, in that the reasoning about C programming can be done quite separately from the reasoning about numerical correctness and numerical accuracy. The tools are *general*, in that they accommodate almost the entire C language (with pointer data structures, function pointers, control flow, etc.) and applied mathematics (reasoned about in a general-purpose logic and proof assistant with substantial libraries for mathematical reasoning). We demonstrate on a simple Newton's-method square root function.

---

1. INTRODUCTION

Formal verifications of functional correctness for programs in real-world imperative languages should be layered:

High-level specification	
	Properties proof: *model satisfies high-level spec.*
Functional model	
	Refinement proof: *program implements model*
Imperative program	

Many authors have described such a layering, more than we can hope to cite.

Ideally, each of these verifications is *machine-checked* (done in a logical framework that can check proofs) and *foundational* (the program logic or other reasoning method is itself proved sound in a logical framework). In many cases, however, one or another of these proofs is done by hand, or left out—typically because of missing tool support for one layer or the other.

And ideally, when machine-checked tools are available for both the properties proof and the refinement proof, they should *connect foundationally*: that is, foundational proofs of the two components should be expressible in the *same* logical framework, so that the composition lemma is just another machine-checked proof in the same framework.

Our contribution in this paper is to demonstrate how very modular such proofs can be, using appropriate foundational verification tools at each layer. In particular, we use the VST tool for C programs, and the Flocq library and Gappa tool for accurate reasoning about numerical algorithms. In addition, we achieve the first end-to-end foundational proof about numerical-methods code in C, and the C code can be further processed by a formally-verified compiler (CompCert) that shares its foundations with VST.

The Verified Software Toolchain [ADH+14] (hereafter referred to as VST, and avaiable at vst.cs.princeton.edu) is a program logic, embedded in the Coq proof assistant, for proving correctness of C programs—for example, that C programs refine functional models. VST has been used in conjunction with properties-proof tools in different application domains (such as the FCF tool in the cryptography domain) to obtain end-to-end (foundationally connected) machine-checked foundational proofs of the correctness of C programs with respect to high-level specifications, using functional programs as the functional models of standard cryptographic algorithms.

Flocq [BM11] (flocq.gforge.inria.fr) is a formalization in Coq of the IEEE 754 floating-point standard. It provides not only a constructive specification of the bit-for-bit representations of sign, exponent, mantissa, etc., but also a theory, a lemma library for reasoning about floating point computations in Coq's logic. Gappa (gappa.gforge.inria.fr) is a tool intended to help verifying and formally proving properties of numerical programs in floating-point or fixed-point arithmetic, using interval arithmetic to bound the "gaps" between lower and upper bounds. Gappa can be used as an automatic tactic in the Coq proof assistant (but can also be used independently of Coq).

Here we will show how these independent tools can be used to make end-to-end foundationally connected functional-correctness proofs of C programs that use floating point. The point of connection—the language of functional models—is functional programs (or relations) in Coq.

In particular, we will show that the functional model is such a strong abstraction boundary that: the VST proof was done by the first author, who knows nothing about how to use Gappa; and the Flocq+Gappa proof was done by the second author, who knows nothing about how to use VST. This "modularity of expertise" is an important consideration in forming teams of verification engineers.

VST's program logic is proved sound, in Coq, with respect to the operational semantics of CompCert Clight [Ler09]. This semantics is also a client of the Flocq interface—CompCert's semantics uses Flocq to characterize the semantics of the machine-language's floating point instructions (parameterized appropriately for each target machine's particular instantiation of IEEE floating point, which Flocq is general enough to permit). Our modular proofs also compose with the correctness proof of CompCert, to get an end-to-end theorem about the correctness and numerical accuracy of the assembly-language program, even though our reasoning is at the source-language level.

Our running example will be a naive[1] implementation of single-precision square

[1]But still fairly efficient: Newton's method doubles the number of accurate mantissa bits in each iteration. Single-precision floating point has 23 mantissa bits, so sqrt_newton(x) should terminate in about $\lceil \log_2 23 \rceil = 5$ iterations for $1 \leq x \leq 4$. When $x \sim 2^k$ it will take $(k+10)/2$ iterations. A more sophisticated square root will start Newton's-method iterations from $y_0 = 2^{\lfloor \log_2(x) \rfloor /2}$, which is calculated in constant time: just take the exponent part of the floating-point number and divide by 2. This kind of reasoning is also possible in Flocq, and in VST's C-language interface to Flocq.

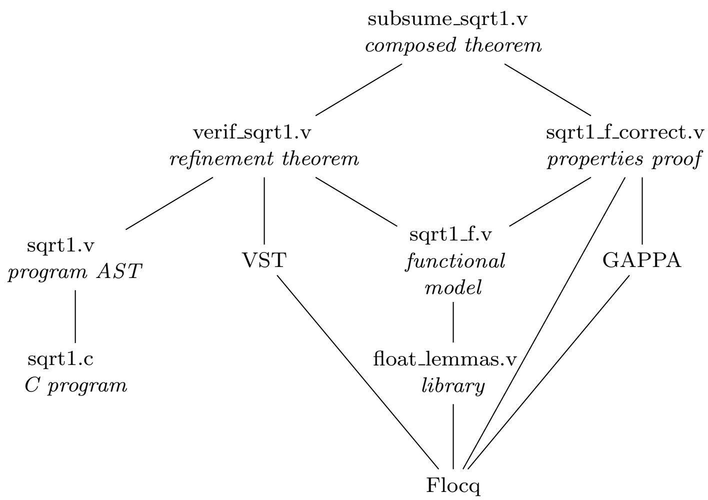

Fig. 1. Module and tool dependency. The .c file is a C program; the .v files are Coq files (definitions and proofs); and the tools and libraries VST, Flocq, and GAPPA are implemented mostly in Coq.

root by Newton's method. Neither of us wrote this program: it was written independently as part of the *Cbench* benchmark suite [vEFG+19], a challenge for implementers of C verification tools. Therefore the program demonstrates, in a small way, that our techniques do not require C programs to be written in a special format, nor synthesized from some other specification.

Theorem. For inputs x between 1 and half the largest floating-point number, sqrt_newton(x) is within a factor of $3 \cdot 2^{-23}$ of the true square root.
Proof. By composing a VST proof and a Gappa+Coq proof, as we will explain.

2. IMPERATIVE PROGRAM, HIGH-LEVEL SPEC, FUNCTIONAL MODEL

In this section we present the C program sqrt1.c and the specification of its correctness, which will be a theorem-statement with the name body_sqrt_newton2. We also show the high-level structure of the proof, as illustrated in Figure 1.

(1) The program sqrt1.c contains a hand-written function sqrt_newton.
(2) The *specification* of that function is written in VST's separation logic as the definition sqrt_newton_spec2.
(3) The *proof* that sqrt_newton satisfies sqrt_newton_spec2 is Lemma body_sqrt_newton2.
That lemma is proved by composing, in a modular way, the following pieces:
(4) A *functional model* of the C program is written, by hand, as the Coq function fsqrt.
(5) An intermediate specification for the C program, called sqrt_newton_spec, which we write to help separate the concerns of "proving things about C programs" and "proving things about floating-point Newton's method," This specification just says that the C program refines the functional model given by fsqrt—but does not say that the functional model computes square roots accurately.

(6) The *proof* that sqrt_newton satisfies sqrt_newton_spec is Lemma body_sqrt_newton. This proof does not use any reasoning about Newton's method or any nontrivial reasoning about floating-point or real numbers.
(7) the *proof* that the functional model computes square roots with a precisely stated accuracy is Lemma fsqrt_correct. the proof of this lemma makes no reference to C code or to VST.
(8) The composition of the main lemmas is the main theorem, body_sqrt_newton2.

All the material is available at github.com/cverified/cbench−vst in directory sqrt. The version described here has the tag `sqrt-publication-2020`.

The imperative program is this C function (in sqrt1.c):

```
float sqrt_newton(float x) {
  float y, z;
  if (x <= 0) return 0;
  y = x >= 1 ? x : 1;
  do { z = y; y = (z + x/z)/2; } while (y < z);
  return y;
}
```

We took this program as given, without alteration, from the Cbench benchmark suite [vEFG+19].

The functional model is this Coq program (in sqrt1_f.v):

```
Definition main_loop_measure (xy : float32 * float32) : nat := float_to_nat (snd xy).
Function main_loop (xy : float32 * float32) {measure main_loop_measure} : float32 :=
    let (x,y) := xy in
    let z := Float32.div (Float32.add y (Float32.div x y)) (float32_of_Z 2) in
    if Float32.cmp Clt z y then main_loop (x, z) else z.
  Proof. ... prove that measure decreases ... Qed.
Definition fsqrt (x: float32) : float32 :=
  if Float32.cmp Cle x (float32_of_Z 0)
  then (float32_of_Z 0)
  else let y := if Float32.cmp Cge x (float32_of_Z 1) then x else float32_of_Z 1 in
       main_loop (x,y).
```

We wrote this functional model by hand, closely following the logic of the C program. But this is a functional program, and easier to reason about. More details about this definition are given in Section 3.1.

The high-level specification (in program verification) is an abstract but mathematically precise claim about *what* the program is supposed to accomplish for the user (but not *how* the code does it). Our high-level spec says that the results are within $3 \cdot 2^{-23}$ of the true square root, provided that the input is not denormalized. Our spec is expressed in Coq (using VST's *funspec* notation in verif_sqrt1.v) as,

Definition sqrt_newton_spec2 :=
 DECLARE _sqrt_newton
 WITH x: float32
 PRE [tfloat]
   PROP ($2^{-122} \leq$ f2real(x) $< 2^{125}$)
   PARAMS (Vsingle x)
   SEP ()
  POST [tfloat]
   PROP (Rabs (f2real (fsqrt x) − sqrt (f2real x)) $\leq 3/(2^{23}) *$ R_sqrt.sqrt (f2real x))
   RETURN (Vsingle (fsqrt x))
   SEP ().

This Definition is really the pair of a C-language identifer _sqrt_newton and a VST function-spec WITH... PRE... POST. The WITH clause binds variables (here, x) visible in both precondition and postcondition. Vsingle injects from single-precision floating-point values into CompCert's val type, which is a discriminated union of integers, pointers, double-precision floats, single-precision floats, etc.

In the precondition, 2^{-122} is the minimum positive normalized single-precision floating-point value, and 2^{125} is half the maximum finite value; we could probably improve (increase) the precondition's upper bound. In the postcondition, 2^{-23} is the value of the least significant bit of a single-precision mantissa, so we prove that the result is accurate within 3 times this value.

The refinement specification is expressed (in verif_sqrt1.v) as,

Definition sqrt_newton_spec :=
  DECLARE _sqrt_newton
  WITH x: float32
  PRE[tfloat] PROP () PARAMS (Vsingle x) SEP ()
  POST[tfloat] PROP () RETURN (Vsingle (fsqrt x)) SEP ().

Lemma body_sqrt_newton: semax_body Vprog Gprog f_sqrt_newton sqrt_newton_spec.

The *refinement theorem* body_sqrt_newton says that the function-body f_sqrt_newton satisifies the specification sqrt_newton_spec. The function body f_sqrt_newton is obtained automatically from the C code. The Vprog and Gprog arguments are provided to describe the global context, which gives the types of any global variables that the function might use, as well as the specifications of any functions that this one might call; but since sqrt_newton does not use global variables and calls no functions, then this theorem can use any Vprog and Gprog, such as the empty context.

The important point about body_sqrt_newton is that neither the theorem-statement nor its proof depends on the *correctness* or *accuracy* of the functional model; we are only proving that the C program implements the functional model, using the fact that C's + operator corresponds to Float32.add, and so on. We do *not* need to know what Float32.add actually does, and we don't have to know why the functional model (Newton's method) works.

The properties theorem is expressed (in sqrt1_f_correct.v) as,

Lemma fsqrt_correct:
 $\forall x,$ $2^{-122} \leq$ f2real(x) $< 2^{125} \rightarrow$
  Rabs (f2real (fsqrt x) − sqrt (f2real x)) $\leq 3/(2^{23}) *$ R_sqrt.sqrt (f2real x).

The important point about fsqrt_correct is that neither the theorem-statement nor its proof depends on any knowledge about the C programming language, or VST's program logic, or even that the C program sqrt_newton exists.

And finally, the main theorem is proved (in subsume_sqrt1.v) by a (fairly) simple composition of those two theorems—the C program satisfies its high-level spec:

Lemma body_sqrt_newton2: semax_body Vprog Gprog f_sqrt_newton sqrt_newton_spec2.

3. DEFINING THE FUNCTIONAL MODEL

The specification sqrt_newton_spec expresses that the C function sqrt_newton returns the value of a Coq function fsqrt. The natural way to encode this Coq function is to follow the structure of the C code and match it practically line per line. Where the C code contains a loop, the Coq function will be recursive. In this case our recursive function is main_loop, shown in Section 2.

3.1 Termination of the loop

Verifiable C is a logic of *partial correctness,* so we do not prove that the C loop terminates. But Coq is a logic of *total functions*, so in defining the fsqrt function we must prove that the main_loop recursion terminates.[2]

One standard way in Coq to prove termination of a recursive function $f(z:\tau)$ is to exhibit a *measure* function, of type $\tau \to \mathbb{N}$, so that the measure decreases on every iteration (and, obviously, cannot go below zero). In this case $\tau = \mathsf{float32} \times \mathsf{float32}$ and the measure function is main_loop_measure. To define the recursive function, we rely on the Function capability of the Coq system, which requires a single argument. Here we use a pair to combine the values of the two variables manipulated in the loop body, which are then named x and y after decomposing this pair. It so happens that our measure depends only on the second part y of this pair but (when using Coq's Function command) the measure-function must take the same argument (xy) as the function main_loop.

Function main_loop(x,y) keeps decreasing y, and y cannot decrease forever. To prove that, we map y into $\mathbb{N}$. We exhibit a function float_to_nat: float32→Nat, and prove a monotonicity theorem, $a < b \;\to\; \mathsf{float\_to\_nat}(a) < \mathsf{float\_to\_nat}(b)$.

This theorem is written formally as follows:

```
Lemma float_to_nat_lt a b :
  float_cmp Integers.Clt a b = true →
    (float_to_nat a < float_to_nat b)%nat.
```

Because Coq functions must be total, float_to_nat must map NaNs and infinities to *something* (we choose 0), but in such cases the premise of the monotonicity theorem

[2]We have several choices in writing a functional model. (1) We can model the program as a function fsqrt : float → float, as we have done here, and then we must prove that fsqrt is a total function, as we do in this section. (2) We can model the program as fsqrtn : $\mathbb{N}$ → float → float, where fsqrtn(k)(x) expresses what will be computed in k iterations. (3) We can model this using a (partial) relation, saying in effect "if the C function terminates, then there exists a return value that is in relation with the input argument." In general, the first approach is most elegant and useful, but if the termination proof were particularly difficult (and not needed) we might choose approach (2) or (3).

would be false, and in proofs about main_loop we maintain the invariant that y is finite.

To understand the construction of float_to_nat, consider that the smallest representable positive floating point number has the form 2^{fmin} where *fmin* is a negative integer that depends on the format. If x is a positive real number representable as a floating point number, then $x = 2^e \times m$ where e and m are integers and $fmin \leq e$. The number $x/2^{fmin} = m^{e-fmin}$ actually is a positive integer. This scheme makes it possible to map all floating point numbers to natural numbers, in a way that respects the order between real values on one side and between natural numbers on the other side.

The largest representable floating point number has the form $2^{fmax} - 2^{fmax-s}$, where s is the number of bits used for the mantissa in the floating point number format. We know that there are less than $2^{fmax-fmin}$ positive real numbers representable as floating point numbers. If we call float_to_nat the function that maps 0 to 2^{fmax}, any positive x of the form $2^e \times m$ to $m \times 2^{e-fmin} + 2^{fmax}$ and any negative x of the form $-2^e \times m$ to $-m \times 2^{e-fmin} + 2^{fmax}$, we see that float_to_nat actually performs an affine transformation with respect to the real number value of floating point numbers, with a positive ratio.

4. THE REFINEMENT PROOF

The refinement theorem (in verif_sqrt1.v) is,

Lemma body_sqrt_newton: semax_body Vprog Gprog f_sqrt_newton sqrt_newton_spec.

This says that, in the global context of assumptions about variables (Vprog) and function-specifications (Gprog), the function-body (f_sqrt_newton) satisfies its function-specification (sqrt_newton_spec). The function-body is produced by using CompCert's parser (and 2 front-end compiler phases) to parse, type-check, and slightly simplify the source code (sqrt1.c) into ASTs of CompCert Clight, a high-level intermediate language that is readable in C.

The proof is written in Coq, using the VST-Floyd proof-automation library [CBG+18]. The use of VST-Floyd is described elsewhere [CBG+18, AC18, ABCD15], and the refinement proof for sqrt_newton is quite straightforward, so we will summarize it only briefly.

The refinement proof is 61 lines of Coq:

Forward symbolic execution (in which each "line" is just a single word, typically forward or entailer!)	22	lines
Loop invariant, loop continue-condition, loop post-condition (do-while loops need all three of these)	9	
Witnesses to instantiate existential quantifiers and WITH clauses	3	
Lines with a single bullet or brace	10	
Proofs about the functional model, mostly fold/unfold/rewrite	17	
Total	61	

VST proofs are not always so simple and easy. If the program uses complex data structures or makes heavy (ab)use of C features such as pointer arithmetic, taking the addresses of structure-fields, casts, or other things that make the program hard to reason about—then the program is indeed harder to reason about, and the proofs will be lengthy. Similarly, when the program does not directly follow the logic of the functional model—if it is "clever" in order to be efficient [App15, section 6])—then the proofs will be lengthy.

But simple programs (such as sqrt_newton) have simple proofs. Furthermore, VST supports modular verification of modular programs, so that each function-body has its own proof, and (overall) proof size therefore scales in proportion to program size.

5. THE PROPERTIES PROOF

The properties of interest are in two parts: first we show that nothing goes wrong (no NaNs or similar exceptional floating point numbers are created), second we show that we do compute a value that makes sense (in this case, a close approximation of the square root).

5.1 From floating point data to real numbers

The final conclusion of our formal proofs concerning the C program is that the returned floating point number represents a specific value within a specific error bound. This statement is essentially expressed using real numbers. For this statement to become available, we first have to show that none of the intermediate computations will produce an exceptional value.

In the loop, the following operations are performed: divide a number by another one, add two numbers together, divide a number by 2. This is represented in our formal development by the following expression.

```
Definition body_exp x y :=
  float_div (float_plus y (float_div x y)) float2.
```

Each of the functions involved here may return an exceptional value (an infinity value or the special value `nan`).

We need more precise reasoning on the range of each of the values to make sure that such an exceptional value does not occur. In practice, the division is safe, and this is proved in two different ways depending on whether the number x is larger than 1 or not.

For instance, when x is larger than 1 we can establish the invariant that y is larger than $\sqrt{x}/2$ and smaller than x. In that case, x/y is larger than 1 and smaller than $2\sqrt{x}$. When x is very large, $2\sqrt{x}$ is significantly smaller, and thus still within range. We can then focus on the sum. If y is smaller than x, then $y + x/y$ is not guaranteed to be smaller than x, but we can now study separately the case where x is larger than 4. In this case, $2\sqrt{x}$ is smaller than x, and we can conclude that if x is smaller than half the maximal representable floating point number, then the sum is within range. On the other hand, if x is smaller than 4 and y is larger than $\sqrt{x}/2$ it is easy to show that the sum is smaller than 8 and thus obviously within the range of floating point number representation.

The reasoning work is actually a little more complex than what is presented in the previous paragraph, because each operation is followed by a rounding process.

So it is not the sum $y + x/y$ that we have to focus on, but $y + r(x/y)$, where $r(x/y)$ is the result of rounding to the correct floating point number, which may actually be larger than x/y. The rounding operations add a few minute values everywhere, so that all portions of reasoning have to be modified to account for these minute values. The difficulty comes from the fact that these values are rounding errors with respect to floating-point representations, the magnitude of which is relative to the value being rounded. Relative magnitudes are confusing for many automated tools for numeric computation, because one cannot reason entirely in linear arithmetic.

In the end, we decompose the range of possible inputs into two cases. In the first case x is between a very small value and 1 and y is between another very small value and 2. In the second case, x is between 1 and a very large value and y is between 1/2 and another very large value. This is expressed by the following two lemmas:

Lemma body_exp_val' x y:
bpow r2 fmin ≤ f2real x < 1 →
bpow r2 (2 − es) ≤ f2real y ≤ 2 →
f2real (body_exp x y) = round' (round' (f2real y + round' (f2real x / f2real y)) / 2).

Lemma body_exp_val x y:
1 ≤ f2real x < / 2 ∗ f2real predf_max → $\frac{1}{2}$ ≤ f2real y ≤ f2real predf_max →
f2real (body_exp x y) = round' (round' (f2real y + round' (f2real x / f2real y)) / 2).

In these lemmas, we see that the real interpretation of the floating point expression is explained in terms of regular real number addition and division, with rounding operations happening after each basic real operation. Once we have established that the inputs x and y are within the ranges specified by these two lemmas, we can be sure that all computation will stay away from exceptional values in the floating point format. We can start reasoning solely about real numbers.

5.2 Reasoning about rounding errors

We have already abstracted away from the C programming language; from this point on, we can start to abstract away from the floating point format. We only need to know that a rounding function is called after each elementary operation and use the mathematical lemmas that bound the difference between the input and the output of this rounding function.

As a way to break down the difficulty, we first study the case where $1 \leq x \leq 4$. We then use some regularity properties of computations with floating point numbers to establish a correspondance between the other ranges and this one. This correspondance will be explained in a later section.

A constant that plays a significant role in our proofs is the *unit in the last place*, usually abbreviated as `ulp`. It corresponds to the distance between two floating point values in the interval under consideration. For an input value x between 1 and 4, $\sqrt{x}$ is between 1 and 2 and the unit in the last place is 2^{-23}. In general, computations about `ulp` have to take into account the change of magnitude in the number being considered, but here for input numbers between 1 and 4 we are sure to work with exactly 2^{-23} for the final result.

A proof then revolves around the following two main facts:

(1) if $y > \sqrt{x} + 16\texttt{ulp}$, then $(y + x/y)/2$ is guaranteed to be smaller than y, even after all the rounding operations,
(2) if $\sqrt{x} - 16\texttt{ulp} < y < \sqrt{x} + 16\texttt{ulp}$ then $(y + x/y)/2$ is guaranteed to be distant from $\sqrt{x}$ by at most $3\texttt{ulp}$, even after all the rounding operations,

So, once the value of y enters the interval $(\sqrt{x} - 3\texttt{ulp}, \sqrt{x} + 3\texttt{ulp})$, we know it will stay in this interval. The value that is ultimately returned will have to be in this interval.

The proof of these two facts deserves a moment of attention, because the method to prove them was first to show that the distance between the rounded computation of $(y + x/y)/2$ and the exact computation was bounded by a very small amount ($\frac{5}{2}\texttt{ulp}$). Then, in the first case we showed that the exact computation was so far below y that even with the errors the decrease had to happen. For the second case, the distance between the exact computation and $\sqrt{x}$ is bounded by an even smaller amount.

For the first part, where we prove a bound on the distance between the computation with rounding and the exact computation, we could benefit from the `gappa` tool [BM17, BFM09]. The text of the question posed to `gappa` is so short it can be exhibited here:[3]

```
@rnd = float< ieee_32, ne >;
{s in [1, 2] /\ e in [-32b-23,3] ->
 (   rnd (rnd ( (s + e) + rnd ((s * s) / (s + e))) / 2)
   - ( ((s + e) + ((s * s) / (s + e))) / 2) ) in [-5b-24,5b-24]}
```

In this text, `s` stands for $\sqrt{x}$, `s * s` stands for x, and `s + e` stands for y, `e` is the current error between y and $\sqrt{x}$.

For the second fact above, we use the following mathematical result:

$$\frac{y - \frac{x}{y}}{2} - \sqrt{x} = \frac{(y - \sqrt{x})^2}{2}$$

If we know $|y - \sqrt{x}|$ to be smaller than $16\texttt{ulp}$, that is 2^{-19}, then the exact computation yields a better approximation, with a distance no more than 2^{-39}, which we grossly over estimate using 2^{-24}. When we add the potential rounding errors, we obtain the result[4] that is the main claim of the paper ($3\texttt{ulp}$), remembering that this results holds for x between 1 and 4.

5.3 Scaling proofs

Once we have obtained the proofs for the input between 1 and 4, we generalize the result to other ranges. The nature of floating point computations makes it possible to view this as a simple scaling of all computations and proofs. When adding two floating point numbers of the same magnitude, the same operation is performed on the mantissa, independently of the magnitude, which is preserved in the result (one has to be careful in the case one adds numbers of opposite sign, but this situation does not occur for our case study). Similar characteristics occur for

[3] After a little processing, this statement becomes theorem from_g_proof in our formal development.
[4] This proof is lemma converge_below_16.

multiplication and division, except that the magnitude of results changes. We must be careful when the result magnitude reaches the limits of the range of representable numbers.

To illustrate this point, let's consider two computations with decimal floating point numbers, with only 3 significant digits. In this case, instead of starting with the range 1 to 4, we would start with the range 1 to 100. We would then want to compare computations with x between 1 and 100 with computations with x between 1 and 10000. For instance, Let us consider the computation as it occurs when $x = 3.97*10$ and $y = 7.37$ on the one hand and $x = 3.97*10^3$ and $y = 7.37*10$ on the other hand.

	$x = 3.97 \times 10$, $y = 7.37$	$x = 3.97 \times 10^3$, $y = 7.37 \times 10$
x/y	5.39	5.39×10
$y + (x/y)$	1.28×10	1.28×10^2
$(y + (x/y))/2$	6.40	6.40×10

We see that in the two columns of this table the same computations are being performed, except for adjustments in the powers of 10. These adjustements work in the following way: if we multiply x by 10^{2k} and y by 10^k, then the same significant digits will appear in all intermediate results, that will be the same in the second column after multiplication of the result in the first column by 10^k. This observation takes into account the behavior of rounding functions.

Now if we transpose this observation to binary floats as they are used in the IEEE 754 standard, this scaling result can be described formally using the following logical statement:[5]

$$\begin{aligned} 1 \leq x \leq 4 &\Rightarrow \\ \frac{\sqrt{x}}{2} \leq y \leq 2\sqrt{x} &\Rightarrow \\ \frac{y \times 2^e + (x \times 2^{2*e})/(y \times 2^e)}{2} &= \frac{y + x/y}{2} \times 2^e \end{aligned}$$

This result essentially explains that the last iteration of the loop in `sqrt_newton` will behave similarly, whether x is between 1 and 4 or in most of the range available in floating point numbers. We only need an extra lemma to explain that if y is larger than $2\sqrt{x}$, $((y + x/y)/2)$ is smaller than y, even after rounding, even when x is outside the $[1, 4]$ interval.[6]

6. COMPOSING THE TWO PROOFS

VST's program logic, Verifiable C, has a notion of *funspec subsumption* [BA19]. That is, funspec A can be proved to imply funspec B, independent of any function-bodies that satisfy A. Suppose

$$\begin{aligned} A &= \mathsf{WITH}\, x.\, \mathsf{PRE}\{Px\}\, \mathsf{POST}\{Qx\} \\ B &= \mathsf{WITH}\, y.\, \mathsf{PRE}\{P'y\}\, \mathsf{POST}\{Q'y\} \end{aligned}$$

Then funspec_sub A B means, $\quad \forall x \exists y.\ P'(y) \rightarrow (P(x) \wedge (Q(x) \rightarrow Q'(y)).$

[5] This is lemma body_exp_scale in the formal development.
[6] Lemma body_exp_decrease16l for $x < 1$ and lemma body_exp_decrease16' for $1 \leq x$.

The logic's *subsumption* rule says that if we have proved that function f satisfies specification A, then it also satisfies B:

$$\frac{\mathsf{semax\_body}\ V\ \Gamma\ f\ A \qquad \mathrm{funspec\_sub}\ A\ B}{\mathsf{semax\_body}\ V\ \Gamma\ f\ B}$$

We have already proved that sqrt_newton satisfies sqrt_newton_spec, that is, the C function implements the Coq function. The theorem fsqrt_correct tells us the properties of the Coq function, so we can use subsumption to give a more informative specification, sqrt_newton_spec2.

The Coq proof of funspec_sub sqrt_newton_spec sqrt_newton_spec2 (in subsume_sqrt1.v) is 12 simple lines of Coq.

7. RELATED WORK

This work is motivated by the desire to provide an answer to a benchmark question on the ability to formally verify C programs. This particular case study concentrates on a C program with numeric computations using floating point numbers. The community of researchers interested in the computation of floating point numbers have a benchmark suite of their own [DMP+16].

Harrison did formal machine-checked proofs of low-level numerical libraries using the HOL-Light system [Har96], based on a formalization of IEEE-754 floating point as implemented on Intel processors—including square root [Har03]. A more abstract and parameterized model of floating-point numbers for HOL-light was developed later [JSG15]—but it is less precise, as it does not include the description of elements known as NaNs (Not a Number).

Russinoff also provided a formal description of floating point technology, but with an objective of producing hardware instead of software [Rus19]. The book also contains descriptions of square root functions.

A previous study of imperative programs computing square roots concentrated on square roots of arbitrary large integers [BMZ02]. The proof was based on the `Correctness` extension of the Coq system [Fil98]. This particular study also involved obligations concerning arrays of small numbers (used to represent arbitrary large integers), so proofs about updates of arrays were needed. However, this study has the same drawback as the one based on Frama-C that the chain between formal proofs and actual executed code is broken: the semantics of the imperative language was only axiomatized and not grounded in a formal language description that is shared with the compiler.

The program we verify here uses Newton's method, which computes the roots of arbitrary differentiable functions. This method was already the object of a formal study in Coq, with the general point of view of finding roots of multivariate functions and Kantorovitch's theorem [PAS11]. That study already included an approach to take rounding errors into consideration, although with an approach that is different from what happens with fixed-point computations. That study did not consider the particular semantics of C programs.

Frama-C [CKK+12] is a verification tool for C programs. It generates verification conditions for C programs; for floating-point programs these can be expressed in terms of Flocq and proved in Coq [BM11]. But Frama-C's program logic is weaker than VST's in three important ways: it is not a separation logic (hence, data structures will be harder to reason about); it is not embedded in a general-purpose logic (hence, the mathematics of the application domain will be harder to reason about); and Frama-C is not foundationally connected to the operational semantics of C (hence, there is no machine-checked proof about the compiled code). For our simple square-root function, the first two of these are irrelevant: sqrt_newton does not use data structures, and the C program proof separates nicely from the application-domain proof. But for nontrivial C programs that use *both* data structures and floating point, and where some aspects of the refinement proof may rely on mathematical properties of the values being represented, VST may have important advantages.

Our work is reminiscent of the work by Boldo et al. on the formal verification of a program to compute the one-dimensional (1D) wave equation [BCF+10]. Their mathematical work is much more substantial, since they reason about the resolution of a partial differential equation. The same work is later complemented with a formal study of the corresponding C program, but they use Frama-C for the last part of the reasoning [BCF+13].

Interval reasoning can often be used to provide formal guarantees about the result of computations, and it is indeed the nature of our final result: the computation is within an interval of ± 3 `ulp` of the mathematical value that we are seeking to compute. Part of the interval reasoning can be done automatically, and we did so with the help of the Gappa tool [BM17, BFM09]. This tool provides proof of interval bounds for some computations and has also been used as a way to guarantee the correctness of other libraries, like the CRlibm library, which promises correct rounding for a large collection of mathematical functions [DDdDM03].

Another attempt to use a general-purpose theorem prover to provide guarantees about floating point computation relies on the PVS system and a static analysis of programs [STF+19]. This tool can then be used to generate code with logical assertion in ACSL to be fed to Frama-C. While this approach provides more automation for the proofs, it still falls short with respect to end-to-end verification.

More related work is described at the *Floating-point Research Tools* page, `https://fpbench.org/community.html`.

8. CONCLUSION

Reasoning about programs is done at many different levels of abstraction: hardware, machine language, assembly language, source code, functional models, numerical methods, and the mathematics of the application domain (which itself may contain levels of abstraction). In formal machine-checked program verification, it is important to separate these different kinds of reasoning. One should use the appropriate theories and tools for each level, and avoid entanglement between tools meant for different levels, and between reasoning methods appropriate for different levels.

Operational semantics is a good abstraction boundary between CompCert's compiler-correctness proof and VST's program-logic soundness proof; and *separation Hoare logic* is a good abstraction boundary between the program-logic soundness proof and the particular program's refinement proof.

We have shown here that the Flocq specification of floating point, combined with ordinary functional programming in Coq's Gallina language, is a good abstraction boundary between C programs and numerical reasoning. The VST refinement proof is concerned with the layout and representation of data structures, with control structures, and (if applicable) concurrency. The Flocq+Gappa numerical-methods proof is concerned with pure functions on floating point numbers, pure functions on real numbers, and the interval-arithmetic reasoning that relates the two. These very different kinds of reasoning are well separated.

But because all of these tools are embedded in Coq, and have foundational soundness proofs in Coq, they connect with an end-to-end theorem *in Coq, with no gaps,* about the behavior of the compiled program, assuming as axioms only the operational semantics of the target-machine assembly language.

ACKNOWLEDGMENTS

We thank Michael Soegtrop for assistance in configuring CompCert, Flocq, and VST. We thank Laurence Rideau and Guillaume Melquiond for their help in navigating the Flocq library. This research was supported in part by National Science Foundation grant CCF-1521602.

References

[ABCD15] Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. *Verifiable C: Applying the Verified Software Toolchain to C Programs*. 2015. `https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf`.

[AC18] Andrew W. Appel and Qinxiang Cao. *Verifiable C*, volume 5β of *Software Foundations*. 2018. `www.cs.princeton.edu:~appel/vc`.

[ADH+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers.* Cambridge, 2014.

[App15] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015.

[BA19] Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. In *23rd International Symposium on Formal Methods*, pages 573–590, 2019.

[BCF+10] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP'10 – Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 147–162, Edinburgh, United Kingdom, July 2010. Springer.

[BCF+13] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *16th Symposium on the Integration of Symbolic Computation and*

Mechanised Reasoning, volume 5625, pages 59–74, Grand Bend, Ontario, Canada, July 2009.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252, 2011.

[BM17] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs.* ISTE Press—Elsevier, December 2017.

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 22(3–4):225–252, 2002.

[CBG+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Automated Reasoning*, 61(1-4):367–422, June 2018.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[DDdDM03] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM: A correctly rounded elementary functions library. In *SPIE 48th annual Meeting International Symposium on Optical Science and TEchnology*, San Diego, USA, August 2003.

[DMP+16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In *Numerical Software Verification, NSV'16*, LNCS. Springer, July 2016.

[Fil98] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *Proceedings of the TYPES'98 workshop*, volume 1657 of *Lecture Notes in Computer Science*. Springer, 1998.

[Har96] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[Har03] John Harrison. Formal verification of square root algorithms. *Formal Methods in System Design*, 22(2):143–153, 2003.

[JSG15] Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. A parameterized floating-point formalizaton in HOL Light. *Electronic Notes in Theoretical Computer Science*, 317:101 – 107, 2015. The Seventh and Eighth International Workshops on Numerical Software Verification (NSV).

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[PAS11] IOANA PASCA. Formal proofs for theoretical properties of Newton's method. *Mathematical Structures in Computer Science*, 21(4):683–714, 2011.

[Rus19] David M. Russinoff. *Formal Verification of Floating-Point Hardware Design, A Mathematical Approach.* Springer, 2019.

[STF+19] Rocco Salvia, Laura Titolo, Marco A. Feliú, Mariano M. Moscato, César A. Muñoz, and Zvonimir Rakamaric. A mixed real and floating-point solver. In Julia Badger and Kristin Yvonne Rozier, editors, *Proceedings of the 11th NASA Formal Methods Symposium (NFM)*, volume 11460 of *Lecture Notes in Computer Science*, pages 363–370. Springer, 2019.

[vEFG+19] Marko C. J. D. van Eekelen, Daniil Frumin, Herman Geuvers, Léon Gondelman, Robbert Krebbers, Marc Schoolderman, Sjaak Smetsers, Freek Verbeek, Benoît Viguier, and Freek Wiedijk. A benchmark for C program verification. *CoRR*, abs/1904.01009, 2019.

4 March 2021: It has come to the authors' attention that citation STF+19 is to the wrong paper; that paper is not connected to Frama-C and does not verify C code. More relevant related work would include the following, both of which use PVS (to relate real-number algorithms to floating-point algorithms) and Frama-C (to relate floating-point algorithms to C code):

Laura Titolo, Mariano M. Moscato, Cesar A. Muñoz, Aaron Dutle, and François Bobot. A Formally Verified Floating-Point Implementation of the Compact Position Reporting Algorithm. In *Formal Methods - 22nd International Symposium,* 2018.

Mariano M. Moscato, Laura Titolo, Marco A. Feliu, and Cesar A. Muñoz. A Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm. In *Formal Methods - 23rd International Symposium,* 2019.