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 foundational**ly: 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.

The Verified Software Toolchain [ADH+14](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. It 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 \cite{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. This semantics is also a client of the Flocq interface, which it uses to characterize the semantics of machine-language floating point (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 \cite{Ler09}, 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 implementation of single-precision square root by Newton’s method. Neither of us wrote this program: it was written independently as part of the Cbench benchmark suite \cite{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, \(\text{sqrt}_0(\text{newton}(x))\) 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.

\footnote{\textsuperscript{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 \(\text{sqrt}_0(\text{newton}(x))\) 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 \(y_0 = 2^{(\log_2 (x))/2}\), 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.}
2. IMPERATIVE PROGRAM, HIGH-LEVEL SPEC, FUNCTIONAL MODEL

All programs, specifications, and proofs are available at github.com/cverified/cbench
in directory sqrt.

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

```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;
}
```

The functional model is this Coq program (in sqrt1.f.v):

```coq
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).
```
The high-level specification is expressed in Coq (using VST’s fanspec 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}) \ast R_sqrt.sqrt (f2real x))
    RETURN (Vsingle (fsqrt x))
    SEP ()
```

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 least significant bit of a single-precision mantissa, so we prove that the result is accurate within 3 times the lsb.

The refinement theorem 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 ().
```

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 `sqrt1f_correct.v`) as,

```
Lemma fsqrt_correct:
  \forall x, \quad 2^{-122} \leq f2real(x) < 2^{125} \rightarrow
  \quad Rabs (f2real (fsqrt x) - sqrt (f2real x)) \leq 3/(2^{23}) \ast 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 end-to-end 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.

One standard way in Coq to prove termination of a recursive function \( f(z : \tau) \) is to exhibit a measure function, of type \( \tau \rightarrow \mathbb{N} \), so that the measure decreases on every iteration (and, obviously, cannot go below zero). In this case \( \tau = \text{float32} \times \text{float32} \) and the measure function is `main_loop_measure`.

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 \rightarrow \text{Nat}`, and prove a monotonicity theorem, \( a < b \rightarrow \text{float_to_nat}(a) < \text{float_to_nat}(b) \).

This theorem is written formally as follows:

Lemma `float_to_nat lt a b`:

\[
\text{float \_ cmp \ Integers.Clt \ a \ b = true } \rightarrow \text{float \_ to \_ nat \ a < float \_ to \_ nat \ b \%nat}.
\]

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 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 \( 2^{f_{\text{min}}} \) where \( f_{\text{min}} \) 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 \( f_{\text{min}} \leq e \). The number \( x/2^{f_{\text{min}}} = m \times 2^{e-f_{\text{min}}} \) 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 \( 2^{f_{\text{max}}} - 2^{f_{\text{max}}-s} \), where \( s \) is the number of bits used for the mantissa in the floating point number format. We know that there are less than \( 2^{f_{\text{max}}-f_{\text{min}}} \) positive real numbers representable as floating point numbers. If we call `float_to_nat` the function that maps 0

\[We have several choices in writing a functional model. (1) We can model the program as a function `fsqrt : float \rightarrow 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 `fsqrt:n \rightarrow float \rightarrow float`, where `fsqrt(n)(x)` expresses what will be computed in \( n \) 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).\]
to $2^{f_{\text{max}}}$, any positive $x$ of the form $2^e \times m$ to $m \times 2^{e-f_{\text{min}}} + 2^{f_{\text{max}}}$ and any negative $x$ of the form $-2^e \times m$ to $-m \times 2^{e-f_{\text{min}}} + 2^{f_{\text{max}}}$, we see that \texttt{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 \texttt{verif_sqrt1.v}) is,

\textbf{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 (tactic+lemma) 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 entail!) | 22 lines |
| Loop invariant, loop continue-condition, loop post-condition (do-while loops need all three of these) | 9 lines |
| Witnesses to instantiate existential quantifiers and WITH clauses | 3 lines |
| Lines with a single bullet or brace | 10 lines |
| Proofs about the functional model, mostly fold/unfold/rewrite | 17 lines |
| **Total** | **61 lines** |

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.
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** \texttt{body.exp.val' x y}:  
\[
\begin{align*}
\text{bpow r2 fmin} \leq \text{f2real x} < 1 & \rightarrow \\
\text{bpow r2 (2 - es)} \leq \text{f2real y} \leq 2 & \rightarrow \\
\text{f2real (body.exp x y)} = \text{round'} \left( \text{round'} \left( \text{f2real y} + \text{round'} \left( \text{f2real x} / \text{f2real y} \right) \right) / 2 \right).
\end{align*}
\]

**Lemma** \texttt{body.exp vals x y}:  
\[
\begin{align*}
1 \leq \text{f2real x} < 2 & \rightarrow \text{f2real predf}_\text{max} \rightarrow \\
\frac{1}{2} \leq \text{f2real y} \leq \text{f2real predf}_\text{max} & \rightarrow \\
\text{f2real (body.exp x y)} = \text{round'} \left( \text{round'} \left( \text{f2real y} + \text{round'} \left( \text{f2real x} / \text{f2real y} \right) \right) / 2 \right).
\end{align*}
\]

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 \(\text{ulp}\). 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 \(\text{ulp}\) 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 \(y > \sqrt{x} + 16\text{ulp}\), then \((y + x/y)/2\) is guaranteed to be smaller than \(y\), even after all the rounding operations,

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

So, once the value of \(y\) enters the interval \((\sqrt{x} - 3\text{ulp}, \sqrt{x} + 3\text{ulp})\), 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 \((52\text{ulp})\). 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:

```plaintext
@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 $\sqrt{x}$, $s \times s$ 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:

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

If we know $|y - \sqrt{x}|$ to be smaller than $16\mathrm{ulp}$, 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 that is the main claim of the paper ($3\mathrm{ulp}$), 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 number of opposite sign, but this situation does not occur for our case study). Similar characteristics occur for 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 $x = 3.97 \times 10$ and $y = 7.37$ on the one hand and $x = 3.97 \times 10^3$ and $y = 7.37 \times 10$ on the other hand.

<table>
<thead>
<tr>
<th></th>
<th>$x = 3.97 \times 10$, $y = 7.37$</th>
<th>$x = 3.97 \times 10^3$, $y = 7.37 \times 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x/y$</td>
<td>5.39</td>
<td>5.39 \times 10</td>
</tr>
<tr>
<td>$y + (x/y)$</td>
<td>1.28 \times 10</td>
<td>1.28 \times 10^2</td>
</tr>
<tr>
<td>$(y + (x/y))/2$</td>
<td>6.40</td>
<td>6.40 \times 10</td>
</tr>
</tbody>
</table>

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 adjustments work in the following way: if we multiply $x$ by $10^2$ 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

\[\text{converge_below_16}\]
statement

\[\begin{align*}
1 \leq x \leq 4 &\Rightarrow \\
\frac{\sqrt{x}}{2} \leq y \leq 2\sqrt{x} &\Rightarrow \\
y \times 2^e + (x \times 2^{2+e})/(y \times 2^e) &\leq \frac{y + x/y}{2} \times 2^e
\end{align*}\]

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. 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

\[
A = \text{WITH} \ x. \ \text{PRE} \ \{P_x\} \ \text{POST} \ \{Q_x\} \quad B = \text{WITH} \ y. \ \text{PRE} \ \{P_y\} \ \text{POST} \ \{Q'_y\}
\]

Then funspec sub \(A \ B\) means, \(\forall x \exists y. \ P'(y) \Rightarrow (P(x) \land (Q(x) \Rightarrow Q'(y))\).

The logic’s subsumption rule says that if we have proved that function \(f\) satisfies specification \(A\), then it also satisfies \(B\):

\[
\frac{\text{semax} \ \text{body} \ V \ \Gamma \ f \ A \quad \text{funspec} \ \text{sub} \ A \ B}{\text{semax} \ \text{body} \ V \ \Gamma \ f \ B}
\]

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.

---

5 This is lemma `body_exp_scale` in the formal development.
6 Lemma `body_exp_decrease16` for \(x < 1\) and lemma `body_exp_decrease16'` for \(1 \leq x\)
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 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 \( \pm 3 \text{ulp} \) 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

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


[DMP+16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard bench-


