VCFloat 2.0 Reference Manual

Andrew W. Appel   Ariel E. Kellison

November 14, 2022

1 Introduction

VCFloat is a tool for Coq proofs about floating-point round-off error. When performing a computation such as $x \times 5.7 + y$ in floating-point with a fixed number of mantissa bits, the result of $x \times 5.7$ cannot always be represented exactly in the same number of bits, ditto the result of the addition $+y$, so some low-order bits must be thrown away—there is *round-off error*.

We can state this more formally with a bit of notation. Let `%F64` be a notation scope in which `*` and `+` and are interpreted as double-precision (64-bit) floating-point operators and `5.7` is interpreted as a double-precision floating-point constant; let R be the function that injects a floating-point value into the reals. Then we might prove that

$$\frac{1 \le x \le 100 \qquad -1000 \le y \le 1000}{|\texttt{(x*5.7+y)\%F64} - (R(x) \times 5.7 + R(y))| \le A}$$

where A is an accuracy bound calculated by VCFloat.

When you prove the correctness and accuracy of a numerical program, there is far more to do than bound the round-off error. If we view `(x*5.7+y)%F64` as a *floating-point functional model* of your program, and $R(x) \times 5.7 + R(y)$ as a *real-valued functional model* of the same program, then the main result of interest can be proved by composing these three theorems:

1. The real-valued functional model finds a solution to the mathematical problem of interest, within accuracy bound A_1.
2. The float-valued functional model approximates the real-valued functional model within accuracy bound A_2.
3. The program (in C or Python or whatever) correctly implements the float-valued functional model.

VCFloat2 provides

- A *modelling language* for describing float-valued functional models and automatically deriving the corresponding real-valued models;
- A prover for bounding roundoff error, the difference between the two models;
- Tools for connecting float-valued models to C programs. But in VCFloat 2.0 (unlike in 1.0) the float-valued modeling language is quite independent of C and can be used to reason about numerical programs in other languages.

To use VCFloat you follow these steps, we will explain below:

1. Write down your floating-point functional model as Coq functions on floating-point values.
2. Pick identifiers for your variables and apply a tactic to reify your model.
3. Automatically derive a real-valued functional model.
4. Specify bounds for your input variables, in a boundsmap.
5. State a roundoff-error theorem, and start the proof using the prove_rndval tactic; this leaves two subgoals, "stage 1" and "stage 2".
6. Prove the stage-1 verification conditions; usually this is as easy as writing all:interval.
7. Prove the stage-2 goal; sometimes this is completely automatic, sometimes you have to assist.
8. (optional) Prove that your C program correctly implements the functional model.
9. (optional) Prove that your real-valued model accurately approximates the mathematical quantity of interest.

2 Floating-point functional models

To use VCFloat you start with,

Require Import vcfloat.VCFloat.

This imports VCFloat's functional modelling language and all of its provers.

Functional models are written as expressions in Coq that apply functions (such as *add* and *multiply*) to variables, constants, and subexpressions that belong to floating-point *types*. We will start with the types.

```
type : Type                    each floating-point format is described as a type
ftype: type → Type              a floating-point number in format t belongs to Coq type ftype(t)
TYPE : ∀(precp : positive) (femax: Z), (fprecp<femax) → (1<fprecp) → type.
Tsingle: type := TYPE 24 128 I I.
Tdouble: type := TYPE 53 1024 I I.
```

That is, you specify a floating-point format, a type, by the number of mantissa bits (e.g., 24 for single-precision, 53 for double-precision, but any number ≥ 2 is legal) and a maximum exponent value (128 for single-precision, 1024 for double-precision, any number greater than the number of mantissa bits). TYPE is a constructor for type, and the I arguments happen to be proofs that 24<128 and 1<24, and so on.

3 NaNs

In the IEEE-754 floating-point standard, one cannot simply *add* two numbers, one must specify how the NaNs will be propagated. That is, if x and y are double-precision floats, what Not-a-Number (NaN) should float-add return if x or y or both are Not-a-Number? Unfortunately that is left to each computer architecture to decide. VCFloat wants to be rigorously faithful to the semantics of the actual computation, so we specify the NaN-propagation behavior of the floating-point model in a typeclass Nans.

The good news is that if your computation never produces any NaNs, then it won't matter which instance of the Nans typeclass you use. And VCFloat helps you prove that your computation never produces NaNs. Then you can parameterize your float-functional-model over a NANS parameter as follows:

```
Section WITHNANS.
Context {NANS: Nans}.

...your functional model goes here
End WITHNANS.
```

That is, the NANS variable can be instantiated with *any* architecture-specific which-nans-to-use structure, and then your float-functional-model will consult this NANS structure whenever it produces a not-a-number, which you will prove is never.

4 Notation Scopes

These notation scopes (and their delimiters) come with VCFloat:

```
Delimit Scope float32_scope with F32.
Delimit Scope float64_scope with F64.
```

Delimiters %F32 and %F64 indicate that constants and operator-symbols should stand for single precision and double-precision (respectively) values and functions.

```
Definition myformula (h: ftype Tdouble) := (5.0e-1 + cast Tsingle (h * 1.6)%F64)%F32.
```

Here, the constant 1.6 and operator $*$ are interpreted in double precision, and the constant 5.0e-1 (which could just as well have been written as .5) and operator + are interpreted in single precision. The variable h is a double-precision floating-point number.

5 Operators

The following operators are available in each notation scope:

$+ \quad - \quad * \quad / \quad < \quad <= \quad > \quad >=$

The minus sign $-$ can be used infix (subtraction) or prefix (negation). The comparison operators can be used in the style $x <= y < z$ as usual in Coq. The following functions can also be used:

BABS (absolute value)

BSQRT (square root)

cast t (cast to ftype(t))

6 Example

A mass on a spring—a harmonic oscillator—with position x and velocity v can be simulated over time-step $h = \frac{1}{32}$ using the Verlet ("leapfrog") method with the formula,

```
Definition h := (1/32)%F32.
Definition F(x: ftype Tsingle) : ftype Tsingle := (3.0−x)%F32.
Definition step (x v: ftype Tsingle) := (x + h∗(v+(h/2)∗F(x)))%F32.
```

Here, the function step is the functional model of (part of) a C program:

```
const float h = 1.0/32.0;
float F (float x) { return 3.0f−x; }
float step (float x, float v) { return x+h∗(v+h/2.0f)∗F(x); }
```

7 Reification

VCFloat will *reify* your functional model into the internal syntax tree that it uses.

To represent *in a logic* a function analyzing logical formulas of type τ, one cannot write a function with type $\tau \to$ Prop; one must operate on *syntactic representations* of floating-point formulas. VCFloat's expr type is for abstract-syntax trees of formulas (you can do Print expr in Coq to see). One can then define in the logic a *reflect* function of type expr $\to \tau$. VCFloat has fval to reflect back to floating-point expressions, and rval to reflect into real-valued formulas. (You can do Check fval or Check rval.)

The opposite process, *reification,* converting from a formula into its abstract-syntax tree, cannot be done *within* the logic, but it can be done by an Ltac program. VCFloat provides the tactic HO_reify_float_expr. One cannot prove a tactic correct, but you do get a per-instance guarantee for each $f : \tau$ by checking that reflect(reify(f)) $= f$.

The reifier will need a *name* for each of your variables. VCFloat's name type is the Coq positive numbers. In our example the variables are x and v, and we will use 1 and 2 for their names:

```
Definition _x : ident := 1%positive. (∗ Variable name for position ∗)
Definition _v : ident := 2%positive. (∗ Variable name for velocity ∗)
```

Here, the Coq variable _x contains not the value, but the *identifier* that we will use for the floating-point variable x. It is not necessary to use consecutive positives, we could have used 5 and 2. Now we can connect _x and _v to x and v as follows:

```
Definition step' := ltac:(let e' := HO_reify_float_expr constr:([_x; _v]) step in exact e').
```

This is a tactical definition of a VCFloat abstract-syntax tree, step', the reified version of step. The tactic is called HO_reify_float_expr, and it expects its second argument (in this case, step) to be a function from (zero or more) floating-point values to a floating-point value. It learns how many arguments there should be from examining the Coq type of step. In this case, since step has type ftype Tsingle $\to$ ftype Tsingle $\to$ ftype Tsingle, the tactic knows that step should have two arguments, both single-precision floats.

The first argument of HO_reify_float_expr should be list of identifiers, to associate with those parameters of the functional model. In this case the list is simpliy [_x;_v].

8 Boundsmap

In order to do round-off analysis one generally needs *bounds* in the input variables: For example, what are the lowest and highest possible values of x and v in our example? We gather information about each variable (name, floating-point type, low-bound, high-bound) into a boundsmap, which maps variable-identifiers to varinfo structures.

Record varinfo := {var_type: type; var_name: ident; var_lobound: R; var_hibound: R}.

To create the boundsmap first make a list of varinfos, then use some ltac boilerplate to compute.

```
Definition step_bmap_list : list varinfo :=
  [ Build_varinfo Tsingle _x 2 4 ; Build_varinfo Tsingle _v (−2) 2 ].

Definition step_bmap : boundsmap :=
   ltac:(let z := compute_PTree (boundsmap_of_list step_bmap_list) in exact z).
```

In the first definition, we make a list of varinfo structures. For each parameter of the functional model, we specify its floating-point precision, its identifier, its lowest possible input value, and its highest possible input value. We put these into a list—in our example, step_bmap_list. Then the tactical definition (step_bmap) is a line of boilerplate that always looks the same (except for the italicized part where you specify this list as shown above).

9 Valmap and reflection

You can *reflect* the abstract-syntax tree (such as step') back into a functional model (such as step). To do that, first make a valmap that relates your variable identifiers to floating-point values.

```
Definition step_vmap_list (x v : ftype Tsingle) := [(_x, existT ftype _ x);(_v, existT ftype _ v)].

Definition step_vmap (x v : ftype Tsingle) : valmap :=
 ltac:(let z := compute_PTree (valmap_of_list (step_vmap_list x v)) in exact z).
```

The auxiliary definition step_vmap_list (when applied to x and v) is a list of pairs, identifier×value, where the "value" is a dependent pair of a type (a floating point format such as Tsingle or Tdouble) and a value of that type. In this case, both x and v are single-precision, but valmaps have the ability to mix precisions.

The second step computes this association list into an efficient data structure.

The function fval evaluates the floating-point interpretation of an AST, in an environment that maps the variables. To *reflect* an AST using a valmap, apply fval as follows:

```
Definition reflected_step (x v: ftype Tsingle) := fval (env_ (step_vmap x v)) step'.

Lemma reflect_reify : ∀x v, reflected_step x v = step x v.
Proof. reflexivity. Qed.
```

The lemma demonstrates that the round-trip—reify then reflect—is indeed the identity function.

10 Real-valued functional model

Suppose we take the float-valued functional model (the step) function) and interpret every constant and operator in the real numbers:

```
Definition step_realmodel' (x v: ftype Tsingle) : R := FT2R x + (1/32)∗(FT2R v + ((1/32)/2)∗(3− FT2R x)).
```

You can make this look prettier using a coercion:

```
Coercion FT2R: ftype >−> R.

Definition step_realmodel (x v: ftype Tsingle) : R := x + (1/32)∗(v + ((1/32)/2)∗(3−x)).
```

In fact, you can automatically derive a real-valued functional model using the rval function, which reflects into the reals much like fval reflects into the floats. Here's a theorem showing that you get what you'd expect:

Lemma correspond_floatmodel_realmodel: ∀x v, rval (env_ (step_vmap x v)) step' = step_realmodel x v.
Proof. intros. unfold step_realmodel. simpl. repeat f_equal; compute; lra. **Qed**.

11 Round-off theorem

The purpose of VCFloat is to prove how accurately the float-valued functional model approximates the real-valued functional model. Here's an example of such a theorem:

Lemma prove_roundoff_bound_step: ∀ *vmap*, prove_roundoff_bound step_bmap *vmap* step' (/ 4000000).

This says, for any valmap *vmap* containing values for x and v that are within the bounds specified by step_bmap, the difference between the floating-point interpretation of step' and the real-number interpretation of step' will be less than one four-millionth.

Recall, of course, that "the floating-point interpretation of step'" is exactly our float-valued functional model; and "the real-number interpretation of step'" is exactly our real-valued functional model.

What if we didn't know the accuracy 1/400000 in advance? VCFloat can calculate it; see Section 13.

Here is how we prove the theorem:

```
Lemma prove_roundoff_bound_step: ∀ vmap, prove_roundoff_bound step_bmap vmap step' (/ 4000000).
Proof.
intros.
prove_roundoff_bound.
-
 prove_rndval.          (* see section 12 *)
 all: interval.
-
 prove_roundoff_bound2.
 prune_terms (cutoff 30).
 do_interval.
Qed.
```

To prove a prove_roundoff_bound theorem, use the prove_roundoff_bound tactic. It leaves two subgoals (delimited by "bullets").

The first subgoal is always proved with prove_rndval, which leaves a few verification conditions. In this case, there are three: proving that the additions and subtractions do not overflow. Usually the subgoals left by prove_rndval are easy to prove using the Coq Interval package, as shown here by all:interval. Section 12 explains and discusses these goals.

The second subgoal is always proved by prove_roundoff_bound2, which leaves one subgoal. In this case the subgoal is,

```
NANS : Nans
v_v : R,        BOUND : −2 ≤ v_v ≤ 2
v_x : R,        BOUND0 : 2 ≤ v_x ≤ 4
e0 : R,        E : Rabs e0 ≤ powerRZ 2 (−150)
d : R,         E0 : Rabs d ≤ powerRZ 2 (−24)
e1 : R,        E1 : Rabs e1 ≤ powerRZ 2 (−150)
e2 : R,        E2 : Rabs e2 ≤ powerRZ 2 (−150)
d0 : R,        E3 : Rabs d0 ≤ powerRZ 2 (−24)
e3 : R,        E4 : Rabs e3 ≤ powerRZ 2 (−150)
e4 : R,        E5 : Rabs e4 ≤ powerRZ 2 (−150)
d1 : R,        E6 : Rabs d1 ≤ powerRZ 2 (−24)
______________________________________(1/1)
Rabs ((v_x + (1/32 * ((v_v + (1/64 * ((3−v_x)*(1+d0)+e1) + e4)) * (1+d) + e3) + e2)) * (1+d1) + e0
       − (v_x + 1/32 * (v_v + 1/32 / 2 * (3−v_x))))
 ≤ / 4000000
```

That is, the real-valued variables v_x and v_v, which represent the values of x and v, are within the bounds specified in the boundsmap. The variables $\delta, \delta_0, \delta_1$ that represent relative errors of additions and subtractions, are each less than 2^{-24} in absolute value. The variables $\epsilon, \epsilon_0, \epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4$ that represent absolute errors of additions and multiplications are each less than 2^{-150} in absolute value. Finally, assuming all of that, one must prove that the difference between the computation *with* all the deltas and epsilons and the computation *without* the deltas and epsilons, is less than the accuracy bound.

To prove this, one can use the Coq Interval package. But in many cases one must do some work to prepare the goal for solution by Interval. Later chapters will explain. In this example, we use the prune_terms tactic. Then the goal solves by do_interval.

12 Verification conditions

The first subgoal left by the prove_roundoff_bound tactic takes the form,

```
prove_rndval bmap vmap step'
```

which encapsulates all the *verification conditions* for evaluating the floating-point expressions. These VCs include that certain subexpressions don't overflow into infinities or NaNs, that certain subexpressions don't underflow into subnormal numbers, or even that certain subexpressions *always* underflow.

The prove_rndval tactic expands the prove_rndval goal into one subgoal per VC. A typical subgoal might be equivalent to this:

```
2 goals
NANS : Nans
v_v, v_x : ftype Tsingle
BOUND : −2 <= FT2R v_v <= 2
BOUND0 : 2 <= FT2R v_x <= 4
e1, d1, e2, e3, d2, e4 : R
H : Rabs e1 <= powerRZ 2 (−150)
H0 : Rabs d1 <= powerRZ 2 (−24)
H1 : Rabs e2 <= powerRZ 2 (−150)
H2 : Rabs e3 <= powerRZ 2 (−150)
H3 : Rabs d2 <= powerRZ 2 (−24)
H4 : Rabs e4 <= powerRZ 2 (−150)
______________________________________(1/2)
0 < powerRZ 2 128 − Rabs (((FT2R v_x + (powerRZ 2 (−5) ∗ FT2R v_v + e4)) ∗ (1 + d2)
                              + e3 + (powerRZ 2 (−11) ∗ − FT2R v_x + e2)) ∗ (1 + d1) + e1)
```

This is basically a proof that the floating point evaluation of our formula on x and v does not overflow. That is, $0 < 2^{128} - |E|$, where E is some function of x and v. E has the same order of magnitude as x and v—both of which are bounded by 4. So it should be *extremely easy* to prove this inequality, and indeed, just calling the interval tactic does the job. It's often the case that *all* the subgoals of the prove_rndval tactic are immediately provable this way.

13 Letting VCFloat calculate the accuracy bound

You can ask VCFloat not only to *prove* an accuracy bound, but to *calculate and prove* the bound if you don't know it in advance.

```
Derive step_b
SuchThat (∀ vmap, prove_roundoff_bound step_bmap vmap step' step_b)
As prove_step_bound.
Proof. ... Qed.
```

Here we are using Coq's Derive command (look it up in the Coq manual). This gives the following initial proof goal:

```
NANS : Nans
step_b := ?Goal : R
______________________________________(1/1)
∀vmap : valmap,
prove_roundoff_bound step_bmap vmap step' step_b
```

To prove this, simply subst step_b to put the unification variable ?Goal below the line, and proceed with exactly the same proof as shown at prove_roundoff_bound_step. Then do_interval will instantiate the bound.

After the **Qed**, there will be two new things in the environment:

```
step_b : R = 4644337115725828 / 18889465931478580854784 +
             5910977010728984 ∗ / 96714065569170333976494080
prove_step_bound: ∀vmap: valmap, prove_roundoff_bound step_bmap vmap step' step_b
```

Displaying bounds in floating point. The bounds derived by the Interval package (via our do_interval tactic) will typically be constructed from rational arithmetic, as shown in the step_b example here. To see this as a floating-point number, you can use the ShowBound tactic.

```
Check ltac:(ShowBound step_b).
(* 2.464803403965517e−7%F64 : Bits.binary64 *)
```

14 field_simplify

The subgoal remaining after prove_roundoff_bound2 has the form, $|E' - E| \leq A$, where E is the real-valued functional model, E' is the real-valued model with deltas and epsilons inserted to represent round-off errors, and A is the accuracy bound. Goals of this kind can be given to the Coq Interval package to solve. Coq.Interval works by computing in interval arithmetic, where every number is represented by a pair of floating-point numbers representing a lower bound and upper bound. For example, to subtract $a-b$ represented as $(a_{\mathrm{lo}}, a_{\mathrm{hi}}) - (b_{\mathrm{lo}}, b_{\mathrm{hi}})$, the result is $(a_{\mathrm{lo}} - b_{\mathrm{hi}},\ a_{\mathrm{hi}} - b_{\mathrm{lo}})$. For soundness Interval even takes care to set the floating-point rounding modes to round down when computing the lo-bound, and round-up when computing the high bound.

When variables such as v_x and v_v can take on a wide range of possible values, Interval uses repeated *bisection* to measure many subranges of v_x and v_v, taking the maximum error.

But there's a problem. Consider the interval calculation of $(a+\delta) - a$, which comes out to

$$(a_{\mathrm{lo}} + \delta_{\mathrm{lo}} - a_{\mathrm{hi}},\quad a_{\mathrm{hi}} + \delta_{\mathrm{lo}} - a_{\mathrm{lo}}).$$

If the value of a is only approximately known, so $a_{\mathrm{hi}} - a_{\mathrm{lo}}$ is large, then the interval approximation of $(a+\delta) - a$ is similarly large—which is bad. But we know that whatever the *true* value of a is, when subtracted from itself it will yield zero. That is, a much better approximation can be obtained by *symbolically* subtracting $a + \delta - a = \delta$ before calling the interval package.

If you examine the proof goal at the end of section 11, you'll see that it contains (more or less) v_x$(1+\delta)-$v_x and v_v$(1+\delta)-$v_v. So we can expect the Interval tactic to perform badly on this expression. The solution is to symbolically simplify the expression, and Coq's field_simplify tactic can do that:

```
match goal with ⊢ Rabs ?a <= _   ⇒ field_simplify a end.
```

This changes the below-the-line portion of the proof goal to,

```
Rabs((−v_x*d0*d*d1−v_x*d0*d−v_x*d0*d1−v_x*d0−v_x*d*d1−v_x*d+2047*v_x*d1+64*v_v*d*d1+
   64*v_v*d+64*v_v*d1+3*d0*d*d1+3*d0*d+3*d0*d1+3*d0+e1*d*d1+e1*d+e1*d1+e1+64*e4*d*d1+
   64*e4*d+64*e4*d1+64*e4+3*d*d1+3*d+64*e3*d1+64*e3+2048*e2*d1+2048*e2+3*d1+2048*e0)/2048)
 <= / 4000000
```

Applying the interval tactic solves this goal immediately.

But field_simplify is not the best tool for this job; we use it here only to illustrate the principle of (automatically) expanding the formula into a multinomial and symbolically canceling terms. There are two problems: (1) the multinomial can have an exponential number of terms, most of which are negligible; and (2) floating-point functional models do not always expand into nice multinomials.

15 prune_terms

The proof goal at the end of the last section had many terms similar to 3*d0*d*d1 where two or more deltas or epsilons are multiplied together. Since the deltas are bounded by 2^{-24} (in single precision) or 2^{-53} (in double precision), and the epsilons are much smaller than that, their product is probably negligible.

The prune_terms tactic expands a formula into a multinomial (like field_simplify) but also deletes (and bounds) negligible terms—you specify a "cutoff" for what you consider neglible. For example, at the section-11 proof goal, one can write,

```
prune_terms (cutoff 30).
```

which expands into a multinomial, cancels terms symbolically, and deletes all terms that can be bounded by 2^{-30}. The result is,

```
Rabs (1 * d1 * v_x + 1/32 * d * v_v + 1/32 * d1 * v_v)
 <= / 4000000 − 5910977010729000 / 9671406556917033397649408
```

In this goal, the number 5910977010729000 / 9671406556917033397649408 is the sum of the bounds of the negligible terms. The goal solves easily by the interval tactic.

For comparison, using (cutoff 50) gives the goal,

```
Rabs (−1/2048 * d0 * v_x + −1/2048 * d * v_x + 2047/2048 * d1 * v_x +
      1/32 * d * v_v + 1/32 * d1 * v_v + 3/2048 * d0 + 3/2048 * d + 3/2048 * d1)
   <= / 4000000 − 5242471455916076 / 20282409603651670423947251286016
```

in which not as many terms have been neglected. But either way, interval solves the goal.

16 error_rewrites

Suppose an expression for the absolute forward error does not expand into a nice multinomial that is tractable for the prune_terms tactic; suppose it is such a large expression that applying the field_simplify+interval tactic causes Coq to crash. An example of such a problem is the carbonGas benchmark from the FPBench benchmark suite:

carbonGas(v):= P+A*(N/v)*(N/v)*(v−N*B)−K*N*T

where A,B,K,N,P,T are constants. Because v appears in the denominator of terms in carbonGas, VCFloat's prune_terms tactic won't simplify the expression enough for interval to produce a decent bound. A tactic that can be used in cases like this is error_rewrites, which will recursively decompose a main proof goal for absolute forward error into subgoals of smaller subexpressions on related terms using the following equalities.

$$
\begin{aligned}
(\tilde{u} - \tilde{v}) - (u - v) &= (\tilde{u} - u) - (\tilde{v} - v) \\
(\tilde{u} + \tilde{v}) - (u + v) &= (\tilde{u} - u) + (\tilde{v} - v) \\
(\tilde{u} * \tilde{v}) - (u * v) &= (\tilde{u} - u) * v + (\tilde{v} - v) * u + (\tilde{u} - u) * (\tilde{v} - v) \\
\frac{u'}{v'} - \frac{u}{v} &= (u' - u) - (v' - v) * \frac{1}{v} * u
\end{aligned}
$$

We write $\tilde{e}$ to denote an expression with deltas (for relative error) and epsilons (for absolute error); so $\tilde{u} - u$ is just the absolute error in computing the formula u in floating-point (rather than in the real numbers).

Consider the expression for the absolute forward error of carbonGas, which looks like

$$\texttt{Rabs}((\tilde{u} * (1 + \delta_2) + \epsilon_7 - \tilde{v}) * (1 + \delta_6) + \epsilon_0 - (u - v)).$$

Applying error_rewrites produces – supposing that $\tilde{u}$, $\tilde{v}$, u, and v are opaque – three subgoals:

subgoal 1: $\texttt{Rabs}(\tilde{u} * (1 + \delta_2) * (1 + \delta_6) + \epsilon_7 - u) \leq ?e3$

subgoal 2: $\texttt{Rabs}(\tilde{v} * (1 + \delta_6) - v) \leq ?e2$

subgoal 3: $\texttt{Rabs}(\epsilon_0) \leq ?e1$

where $?e1, ?e2, ?e3$ are unification variables to be determined in subproofs, and the total error is now bounded by $e3 + e2 + e1$.

The subexpressions in the above subgoals are smaller and contain related terms. For these reasons, using the field_simplify+interval tactic is potentially more tractable. However, in some cases, the recursion stops before it fully decomposes a subexpression into a form that interval can provide a decent bound on. In this case, as long as all rational expressions have been decently decomposed by the division case in error_rewrites, the prune_terms tactic can be used successfully.

There are some cases when error_rewrites causes Coq to crash: on expressions with a large number of operations (approximately 40). This occurs because error_rewrites simply produces too many subgoals. As an example, consider that the carbonGas benchmark has 11 floating-point operations and that error_rewrites produces 125 subgoals (and this is even after error_rewrites has automatically discharged a few!).

17 Abstract *versus* transparent valmap

When you have proved a prove_roundoff_bound theorem as described in the previous sections, then you may want to use that in the proof of other theorems about your floating-point program. Since prove_roundoff_bound quantifies over all valmaps, you can apply it to a particular valmap.

For example, step_vmap defined in Section 9 takes floating-point values x and v, and produces a valmap. That is, step_vmap 3.1 0.7 is the valmap with position 3.1 and velocity 0.7. You could apply the prove_roundoff_bound_step theorem to this valmap to prove that, when the program is run with arguments 3.1 and 0.7, then its roundoff error will be less than 1/4000000.

In that theorem, the *vmap* is a quantified variable—it is abstract. But you can also prove a prove_roundoff_bound theorem with a less-abstract vmap.

For example, a theorem like this one (and see Test/summation.v for another example):

Lemma prove_rndoff' :
  ∀ (x v : ftype Tsingle),
  let accuracy := *some function of x and v*
  my_extra_constraint x v →
  prove_roundoff_bound step_bmap (step_vmap x v) step' accuracy.

This example illustrates another thing as well: Suppose you have extra constraints on your variables, besides just the boundsmap. You can state these constraints as an extra hypothesis in the theorem, and the proof tactics (such as interval) can make use of them. You can also prove a theorem in which the accuracy bound is dependent on the variables in your valmap—for example, a *relative error* bound.

18 Annotations

Floating-point error analysis can be slightly more precise in certain cases:

Denorm: When the result of a calculation is known to be a *denormal* (also called *subnormal*) number—a tiny number within $2^{e_{\min}}$ of zero—then it has only an additive error. That is, $(a+b)+\epsilon$ instead of $(a+b)(1+\delta)+\epsilon$.

Norm: When the result of a calculation is known to be a *normal* number—that is, bounded away from zero by at least $2^{e_{\min}}$—then it has only a relative error. That is, $(a+b)(1+\delta)$ instead of $(a+b)(1+\delta)+\epsilon$.

Sterbenz: When a,b satisfy $\frac{1}{2}<\frac{a}{b}<2$, then the floating point subtraction $a-b$ is exact, no relative error δ, no absolute error ϵ.

You can annotate these cases in your functional model using these functions:

```
Definition Norm {A}(x: A) := x.
Definition Denorm {A}(x: A) := x.
Definition Sterbenz {A}(x: A) := x.
```

As you can see, these are just identity functions, so semantically they do nothing. But they guide VCFloat's reifier to mark its internal abstract-syntax tree. This will cause additional proof obligations (subgoals) at stage 1, to *prove* that such-and-such a subexpression is normal, or denormal, or Sterbenz; but will cause fewer deltas and epsilons to be generated at stage 2.

In our running example we could write,

```
Definition h := (1/32)%F32.
Definition F(x: ftype Tsingle) : ftype Tsingle := Sterbenz(3.0−x)%F32.
Definition step (x v: ftype Tsingle) := Norm(x + h∗(v+(h/2)∗F(x)))%F32.
```

19 Verified Software Toolchain

You can use the Verified Software Toolchain (VST) to prove that a C program correctly implements a floating-point functional model.

Along with importing VST.floyd.proofauto and the other standard boilerplate that introduces a VST proof, you will want:

```
From vcfloat Require Import FPCompCert Float_notations.
Require Import float_model. (∗ your functional model ∗)
```

It is not necessary to import all of vcfloat.VCFloat; the imports shown are enough to connect CompCert's definitions for floating point to VCFloat's definitions. CompCert and VCFloat use different names for the same underlying Flocq floating-point types:

```
Eval compute in compcert.lib.Floats.float32. (∗ = Binary.binary_float 24 128 ∗)
Eval compute in ftype Tsingle.               (∗ = Binary.binary_float 24 128 ∗)
Eval compute in compcert.lib.Floats.float.   (∗ = Binary.binary_float 53 1024 ∗)
Eval compute in ftype Tdouble.               (∗ = Binary.binary_float 53 1024 ∗)
```

In your VST assertions (funspecs, loop invariants, etc.), use the VCFloat names for those types. For example, here we write ftype Tsingle instead of float32:

```
Definition force_spec :=
 DECLARE _force
 WITH q : ftype Tsingle
 PRE [ tfloat ] PROP() PARAMS(Vsingle q) SEP()
 POST [ tfloat ] PROP() RETURN (Vsingle (F q)) SEP().
```

There are two useful tactics to convert CompCert-style float notations to VCFloat-style notations. Immediately after start_function you can write,

```
start_function.
subst MORE_COMMANDS; unfold abbreviate; canonicalize_float_constants.
```

The tactic canonicalize_float_constants converts all of the floating-point literals in the AST of your function-body into a VCFloat style, which makes them easier to reason about (and to relate to your functional model).

The following tactic is useful *after* going forward through a sequence of C statements that perform floating-point operations:

```
autorewrite with float_elim in *.
```

It converts Float32.add x y to $(x+y)$%F32, and similarly for other operators.

That's it! Other than these conversions, you use VST in a completely standard way.

Exactly matching the functional model

When using VST or any other tool to prove that a program correctly implements a functional model, take care to match it exactly. For example, in floating point the associative law $(a+b)+c = a+(b+c)$ does not hold.

The commutative law $a + b = b + a$ may hold *only if neither a nor b is a NaN*, depending on how your target machine propagates NaNs. Therefore, if your C program computes $a + b$ while the functional model computes $b + a$, you will be able to prove it correct only if you propagate the invariant that a is finite and b is finite. It's certainly possible to propagate such invariants, but it's simpler if you don't have to.

20 Examples

VCFloat comes with several worked examples, in the Test and FPBench directories:

Test/TestRefman.v the running example from this reference manual

Test/TestPaper.v the (similar) running example from "VCFloat2: Floating-point error analysis in Coq"

Test/summation.v An example of a relative error bound (this example is still a work in progress)

FPBench/*.v examples from the FPBench benchmark suite (fpbench.org)

21 Bibliography

VCFloat 1.0 was built in 2015 and described in,

A unified Coq framework for verifying C programs with floating-point computations,

by Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin, in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP'16)*, pages 15–26, 2016 (https://doi.org/10.1145/2854065.2854066).

VCFloat 2.0 was built 2021-2022 and described in,
VCFloat2: Floating-point error analysis in Coq, by Andrew W. Appel and Ariel E. Kellison, October 2022 (distributed as doc/vcfloat2.pdf in the vcfloat repo).

VCFloat 2.0 is applied and demonstrated in,
Verified numerical methods for ordinary differential equations, by Ariel E. Kellison and Andrew W. Appel, in *15th International Workshop on Numerical Software Verification (NSV'22)*, August 2022.