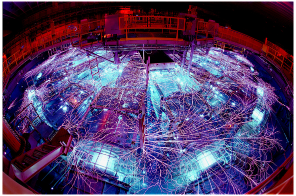
Exceptional service in the national interest



Formal and Semi-Formal Verification of Floating-Point Computations in C Programs

Samuel D. Pollard, Sandia National Labs, Livermore, California

FP Bench Community Meeting, 6 April 2023

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND No. SAND2023-01442PE

Introduction

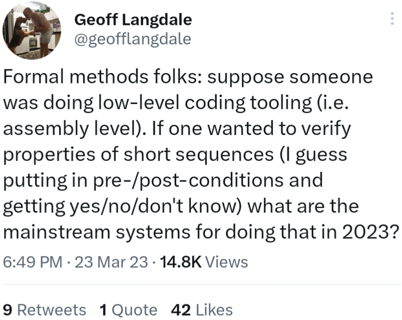
- Sandia National Labs is a US government research & development center
- Sandia does many things, for example
  - we develop high-consequence embedded control systems
  - also a large consumer of HPC for modeling and simulation
- The former is good use case for formal methods
- For example:
  - Kalman Filter
  - Problem size $\sim 20 \times 20$ matrices
  - Implemented in C with minimal dependencies

Methodology, or, What Does "Formal" Mean Anyway?

- When costs for failure are catastrophic, testing is insufficient

So now what? Methods like

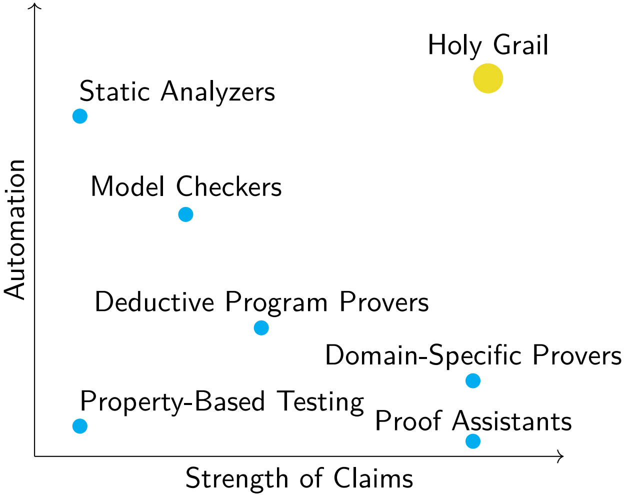
- Uncertainty Quantification (statistical)
- Modeling and Simulation (physics-based)
- Model-Based Systems Engineering (state machines)
- *Formal* Methods
    - Have: English specifications, developers' brains, source code
    - Goal: make a perfect model of a digital system
    - Tools: In my opinion, the biggest barrier to FM adoption



Challenges with Proving Code Correct

In general it's undecidable. But to be more specific ...

1. Formal Models of Hardware
   - ISAs are *usually* a good abstraction to "stop" at
   - Sometimes require axiomatization of hardware (e.g. memory-mapped I/O)
   - Subtle differences between architectures
2. Tooling
   - How strong of assurance do you want?



Adapted from Leroy [5]

Models of Hardware

- Even CompCert gets it wrong sometimes[1]
- Our early analysis found an issue with NaN propagation for RISC-V

```c
// This signaling nan should not be propagated because
// FCVT.S.D or FCVT.D.S (type casting) is _not_ one of the
// operations which preserves NaN payload
float x = single_of_bits(0xFF800009);
printf("FCVT.D.S snan(-9)    = 0x%016llX = 0x7FF8000000000000\n",
     bits_of_double((double) x));
```

```
$ ccomp -interp -quiet nan.c
FCVT.D.S snan(-9)    = 0xFFF8000120000000 = 0x7FF8000000000000
                       ^    observed    ^   ^    correct     ^
```

[1] https://github.com/AbsInt/CompCert/issues/428

Risc-V Standard

Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN. The canonical NaN has a positive sign and all significand bits clear except the MSB, a.k.a. the quiet bit. For single-precision floating-point, this corresponds to the pattern `0x7fc00000`.

We considered propagating NaN payloads, as is recommended by the standard, but this decision would have increased hardware cost. Moreover, since this feature is optional in the standard, it cannot be used in portable code.

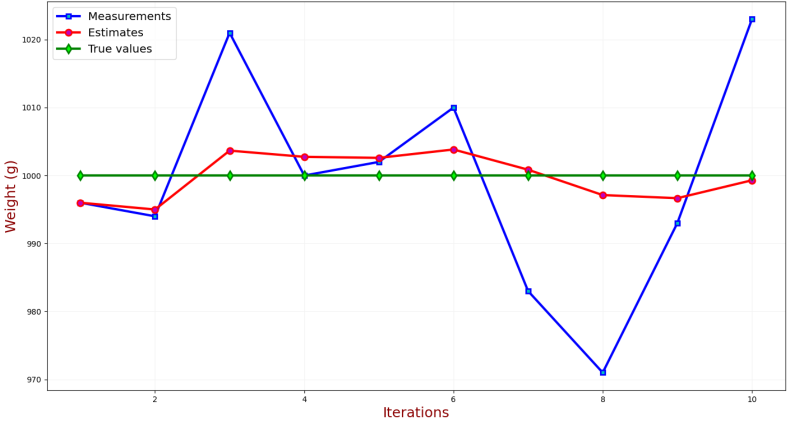
Implementors are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.

RISC-V Manual 2019, Sec 11.3

Might be a good intern project to search for more discrepancies

Application: Kalman Filter

- Start with a simplified case: measuring the weight of a bar of gold with a noisy scale
- Kalman filter assumes noise normally distributed



[2]

First Steps towards a Formally-Verified Kalman Filter

Three values to update

- K Kalman gain, $\in [0, 1]$; how much to prefer new measurements to best guess
- curr current estimation
- p_var monotonically nonincreasing

```c
int main() {
  double input, K;
  double curr = 990.0; // Initial guess
  double p_var = 100.0; // Estimation variance 10^2
  double r_var = 225.0; // Measurement variance 15^2
  while(scanf("%lf", &input) != EOF) {
    K = update_gain(p_var, r_var);
    curr = update_state(K, curr, input);
    p_var = update_p(K, p_var);
  }
  return 0;
}
```

Writing the Spec with VST

- Write the functional spec in Coq
- Challenge: VST does not support `printf`, `scanf` because of varargs

```
Definition update_p (K p : float ) : float :=
  ((Float.of_int (Int.repr 1) - K) * p)%F64.
Definition update_state (K x m : float ) : float :=
  (x + K * (m - x))%F64.
Definition update_gain (p r : float ) : float :=
  (p / (p + r))%F64.
(* Other specs with the main function *)
Definition Gprog := [
  update_p_spec;
  update_state_spec;
  update_gain_spec;
  main_spec].
```

Abbreviated Proof

- Rewrote to not use `scanf`
- 21 lines of C code
- 211 lines of Coq proof

```
Lemma body_main:
  semax_body Vprog Gprog f_main main_spec.
Proof.
  start_function .
  repeat forward.
  forward_for_simple_bound
    10
    (EX i: Z,
      (PROP (0 ≤ i ≤ 10)
        LOCAL( (* Local variables *) )
        SEP( (* SEP = Separation Logic (predicates) *) )
      )
    ).
  - (* ...  Proof for the rest of the steps *)
```

Enter VCFloat

- The C program implements the spec, also written in floating point
- But what about floating-point error?
- A realistic workflow is to write the real-number spec, then analyse various types of error of the implementation (dicretization, roundoff)
- VCFloat allows reasoning about this, and proves C implements real spec, within error bound

```
Definition filter_step
  (i : nat) (g : R) (m : R) : R :=
  g + (1 / i * (m − g)).

Fixpoint alpha_filter
    (guess : R) (i : nat) (ns : list R)
    : R :=
  match ns with
  | [] ⇒ guess
  | n:: ns' ⇒
    let i' := (i + 1)%nat in
    let guess' := filter_step i' guess n in
    alpha_filter guess' i' ns'
  end.
```

VCFloat2 [1], `https://github.com/ak-2485/Kalman_Filter`

VCFloat Experience

- Coq specification around same length as C program (30 lines)
- Coq proof of program along with error bounds: 700 lines
- Easier to reason about real numbers instead of floats
  - Typically separate overflow/NaN from the finite case
  - Often requires proofs using Flocq, or adapting Coq library
  - For example, ODEs and their integrators [4]
- But you can get properties parametric in iteration count
- VCFloat helps discover error bound with some modularity, but still requires manual effort

What about Frama-C?

- VCFloat, VST you spend lots of time in the details
- Brittle to code changes
- Is there a less formal way?
- Frama-C [3] parses C along with specs (as preconditions)
- Write specs in ACSL, ANSI C Specification Language
- Dispatch to provers or SMT solvers

Frama-C and Floating Point

- Has a back-end for Gappa
- But I've not had much success with it :(
- This is about the max complexity
- Notice it says nothing about error

```
// True bound: <= 1732.0508075688772935274463415058722
/*@
  requires \valid_read(a);
  requires valid_vect(a); // all elements finite
  requires -1000.0 <= a->v.i && a->v.i <= 1000.0;
  requires -1000.0 <= a->v.j && a->v.j <= 1000.0;
  requires -1000.0 <= a->v.k && a->v.k <= 1000.0;
  ensures \is_finite(\result);
  ensures 0.0 <= \result <= 1732.051;
*/
double norm(Vector_3* a)
{
  return sqrt(a->v.i*a->v.i + a->v.j*a->v.j +
    a->v.k*a->v.k);
}
```

What I'd Really Like

- An example from the Frama-C Manual
- But fails, does not know about `round_error` and `exact`
- Sylvie Boldo et al. have results, but required Coq proofs (may also have bit-rotted)

```
/*@
  requires \abs(\exact(x)) <= 0x1p-5;
  requires \round_error (x) <= 0x1p-20;
  ensures \abs(\exact(\result) - \cos(\exact(x)))
          <= 0x1p-24;
  ensures \round_error(\result)
          <= \round_error(x) + 0x3p-24;
@*/
float cosine(float x) {
  return 1.0f - x * x * 0.5f;
}
```

Frama-C with No FP Still Useful!

- This is why I like Frama-C: incremental, modular FM
- Null pointer exceptions still easy in C...
- Start with a basic spec, move to all callers
- Have already found bugs using this method
- Are we white glove testers? Can we start charging $2,000/hour consulting???

```c
/*@
  requires \valid_read(a);
  requires \valid_read(b);
  requires \valid(C);
  assigns *C;
*/
void axb(const Matrix_MxN* a,
         const Matrix_MxN* b,
         Matrix_MxN* C);
```

Future Work

- Verify more and more of the Kalman Filter
- It's actually an extended Kalman Filter
  - Can't assume normal distribution on errors, measurement variables
- Better tools for C code verification
- formalization that valid assumptions mean this is the optimal filter for our cases
- probability $+$ Coq $=$ scary, need to look into this more

Future Work (Cont.)

- Back to our application: Extended Kalman Filter
- Straightforward implementation easier to reason about
- Straightforward impl takes 85% of runtime
- Looking for verifiable transformations that improve performance

```
void axb(const Matrix_MxN* a,
         const Matrix_MxN* b,
         Matrix_MxN* C) {
  int r, c, i;
  Matrix_MxN ret = {a->m, b->n};
  for (r=0; r<a->m; r++) {
    for (c=0; c<b->n; c++) {
      ret.a[r][c] = 0.0;
      for (i=0; i<a->n; i++)
        ret.a[r][c] += a->a[r][i]*b->a[i][c];
    }
  }
  *C = ret;
}
```

References I

[1] Appel, A. W., and Kellison, A. E.
Vcfloat2: Floating-point error analysis in coq, 2022.
In Submission.

[2] Becker, A.
The α-β-γ filter, 2023.
Available at https://www.kalmanfilter.net/alphabeta.html.

[3] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B.
Frama-c.
In *Software Engineering and Formal Methods* (Thessaloniki, Greece, Oct. 2012), SEFM (LNCS 7504), Springer, pp. 233–247.

[4] Kellison, A. E., and Appel, A. W.
Verified numerical methods for ordinary differential equations.
In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems* (Cham, 2022), O. Isac, R. Ivanov, G. Katz, N. Narodytska, and L. Nenzi, Eds., Springer International Publishing, pp. 147–163.

[5] Leroy, X.
In search of software perfection.
Available at https://youtu.be/lAU5hx_3xRc, Nov. 2016.