Verification of Floating-Point Software

D. N. Hoover
Odyssey Research Associates, Ithaca NY

Abstract

Floating point computation presents a number of problems for formal verification. Should one treat the actual details of floating point operations, or accept them as imprecisely defined? or should one ignore round-off error altogether, and behave as if floating point operations are perfectly accurate? There is the further problem that a numerical algorithm usually only approximately computes some mathematical function, and we often do not know just how good the approximation is, even in the absence of round-off error.

ORA has developed a theory of asymptotic correctness which allows one to verify floating point software with a minimum entanglement in these problems. We describe this theory and its implementation in the Ariel C verification system, also developed at ORA. We illustrate the theory using a simple program which finds a zero of a given function by bisection.

Verification of Floating–Point Software

Douglas Hoover

Odyssey Research Associates, Inc.

Difficulties

- Machine real arithmetic does not have nice mathematical properties
- Doesn't match ideal arithmetic (overflow, round-off, underflow)
- Programs don't satisfy the specification we'd like them to

Asymptotic Correctness

- Specify "ideal behavior" of the program (e.g. "program computes the square root of its input")
- Verify that if program is run on a sequence of machines converging to perfect accuracy, then program's behavior converges to ideal behavior

Advantages of the Asymptotic Approach

- Machine real arithmetic can be specified loosely
- Specifications can be written in terms of ideal behavior
- Verification does not require roundoff error analysis
- Verifies *logical* correctness — absence of "bugs" from inaccuracy of machine arithmetic that are not related to error *magnitude*.

Nonstandard analysis

$$\mathbf{R} \subseteq {}^*\mathbf{R}$$

Standard part map

$$st : {}^*\mathbf{R} \to \mathbf{R}$$

rounds off a finite nonstandard real to an infinitely close standard real.

Continuity

f is continuous at $(a_1, \ldots, a_n)$ if

$$st(f(a_1, \ldots, a_n)) = f(st(a_1), \ldots, st(a_n))$$

Differentiation by algebraic manipulation

Let $st(\epsilon) = 0$, $\epsilon \neq 0$. For all standard x,

$$\begin{aligned} \frac{d(x^2)}{dx} &= st\left(\frac{(x+\epsilon)^2 - x^2}{\epsilon}\right) \\ &= st\left(\frac{2\epsilon x + \epsilon^2}{\epsilon}\right) \\ &= st(2x + \epsilon) \\ &= 2x \end{aligned}$$

Nonstandard Analysis

- Asymptotic approach can be formalized naturally in nonstandard analysis using infinitesimals
- Primitive operations are assumed to return values which are infinitely close to the ideal values when the arguments and ideal answers are finite
- Programs are specified to have behaviors infinitely close to ideal behavior when inputs are finite

Finding Roots of a Continuous Function

- `find_zero` searchs for a root of a user–supplied function F by bisection.
- At each iteration, it tests to see if the values of F at the left endpoint and the midpoint are of opposite sign, and changes one of the endpoints to the midpoint so as to keep a root between the two endpoints.
- The program terminates when it finds a root or when it reachs a user–supplied bound on the number of iterations.

```
float find_zero(left0,right0,maxit)
float left0,right0;
int maxit;
{
    float left,right,center;
    float cval,lval0,rval0;
    int numit;

    numit = 0;

    lval0 = F(left0);
    rval0 = F(right0);

    left = left0;
    right = right0;
    center = (left + right)/2.0;
    cval = F(center);

    while(cval != 0.0 && numit < maxit) {
        if (lval0 * cval < 0)
            right = center;
        else
            left = center;
        center = (left + right)/2.0;
        cval = F(center);
        lval0 = F(left);
        numit = numit + 1;
    }

    return(center);
}
```

Specification of `find_zero`

IF F is continuous and `find_zero` is started up with

- `left0` and `right0` not "large";
- `maxit` "large";
- F(`left0`) and F(`right0`) of opposite sign

THEN `find_zero` terminates normally (i.e. without an exception) and the value output is "close to" some zero of F.

Attempted Verification

- Proof of termination is easy.
- Proof that termination is normal is a bit harder. Must prove that no overflow happens. To prove this, must prove that the values of the end-points stay in some range of numbers which are not "large".

How would we prove that the program returns an approximation to a root?

- Prove when the program terminates, the endpoints are "close". This follows from the fact that the program halves the interval a "large" number of times.
- Prove there's always a root between the endpoints. This should follow from the way the program decides whether to move the left endpoint or the right. From this we'd get `center` "close to" a root.
  Unfortunately, it's not true that there's always a root between the endpoints.

The Bug

- In the test statement, can have `lval0` and `cval` of opposite sign, but have the product underflow to 0. This causes the program to move the wrong endpoint.
- Tests bear out this bug.

Possible Fixes

Several ways to fix this bug

- Change test to

```
(lval0 < 0 && cval >= 0) ||
(lval0 >= 0 && cval < 0)
```

- Change test so instead of always testing left endpoint against midpoint, it always tests the endpoint with the larger value of F against the midpoint.
  This doesn't necessarily keep a root between the endpoints, but it delivers an approximation to a root anyway.

Ariel

- Verification system for subset of C including real arithmetic and some UNIX system calls.
- Implements nonstandard formalization of the asymptotic approach.

Semantic Verification

- Ariel verifies programs by generating a description of the program's denotation in a higher–order language (the *Clio metalanguage*)
- Specifications are statements about the denotation in the Clio metalanguage
- Verification is a proof of the specification directly from the description of the denotation in Clio theorem prover
- Specifications can be any statement about the program's denotation which can be expressed in the Clio, including termination

C Semantics

- A "run" of the program is modeled as a sequence of *events*
- Events are:
  - the event of going into a certain state
  - terminating and returning a value
  - terminating and returning no value
  - raising an exception
  - an "unknown" event
- The semantics of the program is expressed as a collection of axioms saying which sequences of events can happen in the course of executing the program.

Sample Verifications

- ZBRENT — a program which finds zeros of a continuous function by bisection
- SWAP — a very simple program to swap the contents of 2 locations which contains a surprising bug
- HOSTILE BOOSTER — a suite of programs, developed by Applied Technology Associates for SDIO, that estimate hostile booster trajectories. This verification is currently in progress.
- SECURE DEVICE DRIVER — specification and verification of security for an Ethernet device driver. Currently in progress.