Proof Assistants
Floating-Point Arithmetic and Verification

Guillaume Melquiond

INRIA Saclay – Île-de-France
Laboratoire de Recherche en Informatique

2011-02-15

Outline

1. Floating-Point Arithmetic and Programs
    - Number Representation
    - Rounded Computations
    - Verifying Floating-Point Algorithms
2. Floating-Point Arithmetic and Proofs
    - Interval Arithmetic
    - Proving Mathematical Theorems

Computers and Number Representation

- 32-bit integers with 2-complement sign:
  - $1 + 1 \to 2$,
  - $2147483647 + 1 \to -2147483648$,
  - $100000^2 \to 1410065408$,
  - $-2147483648 \mod -1 \to$ BOOM (floating-point exception?!)
- 64-bit binary floating-point numbers (IEEE-754):
  - $2 \times 2 \times \cdots \times 2 \to +\infty$,
  - $1 \div 0 \to +\infty$,
  - $1 \div -0 \to -\infty$,
  - $0 \div 0 \to$ NaN.

Floating-Point Numbers

Represented by sign s, mantissa m (aka significand), and exponent e:

$$f = (-1)^s \cdot m \cdot \beta^e.$$

Radix β is fixed, usually 2 or 10.

Finite datatype:

- m is a bounded integer (e.g., $|m| < 2^{53}$), limited precision,
- e is a bounded integer (e.g., $-1074 \leq e \leq 970$), limited range.

Consequences: inaccurate results and exceptional behaviors.

Rounded Computations

Value 0.1 cannot be represented as $m \cdot 2^e$.
Closest floating-point number with $|m| < 2^{24}$:

$$0.1 \simeq 13421773 \cdot 2^{-27} = 0.100000001490116119384765625$$

Example

Accumulate 0.1 during 864000 iterations:

```
float f = 0;
for (int i = 0; i < 10 * 60 * 60 * 24; ++i)
  f = f + 0.1f;
printf("f␣=␣%g\n", f);
```

Computed result: $f = 87145.8$. Expected result: 86400. Error: $+0.86\%$

First Gulf War, a Patriot antimissile system has been running for 48 hours, it fails to intercept and destroy a Scud missile: 28 casualties.

Some Other Numerical Failures

- 1983, truncation while computing an index of Vancouver Stock Exchange causes it to drop to half its value.
- 1987, the inflation in UK is computed with a rounding error: pensions are off by £100M for 21 months.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
- 2007, Excel displays the result of 77.1×850 as 100000.
- 2010, PHP servers enter an infinite loop on some decimal inputs.

Verifying Numerical Algorithms

- 2007, Excel displays the result of 77.1×850 as 100000.

  Bug in binary/decimal conversion.
  Failing inputs: 12 FP numbers.
  Probability to uncover them by random testing: 10^{-18}.

Numerical algorithms require detailed proofs of correctness.

But these proofs are long, tedious, and error-prone.
Hence the need for formal methods.

Roundoff and Method Errors

Example (Computing Cosine Around Zero)

```
/*@ requires \abs(x) <= 0x1p-5 ;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23; */
float toy_cos(float x) {
  //@ assert \abs(1.0-x*x*0.5 - \cos(x)) <= 0x1p-24;
  return 1.f - x * x * 0.5f;
}
```

Two kinds of error occur:

- roundoff error between `toy_cos`(x) and $1 - x^2/2$,
- method error between $1 - x^2/2$ and $\cos(x)$.

Note: Method errors cannot be detected without a specification!

Peculiarities of Floating-Point Algorithms

Example (Veltkamp's Algorithm)

```
void split(double x, double *xh, double *xl) {
  double t = 0x8000001 * x;
  *xh = (x - t) + t;
  *xl = x - *xh;
}
```

Replacing floating-point operations by the corresponding real operators does not hint at what the algorithm computes (`*xh` $\not\simeq x$ and `*xl` $\not\simeq 0$).

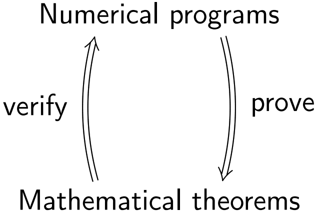
Approaches like abstract interpretation cannot work on this kind of code.

Peculiarities of Floating-Point Algorithms

Example (Dekker's Algorithm)

```
/*@ requires ...
  @ ensures  *zh + *zl = x * y; */
void mul(double x, double y, double *zh, double *zl) {
  double xh, xl, yh, yl;
  split(x, &xh, &xl);
  split(y, &yh, &yl);
  *zh = x * y;
  *zl = -*zh + xh * yh + xh * yl + xl * yh + xl * yl;
}
```

Proving Mathematical Theorems



Numerical programs

verify

prove

Mathematical theorems

While programs are formally verified by theorems,
theorems can be proved by computations (reflection, extraction, etc).

Interval Arithmetic

An expression is overapproximated by a set containing its value.
Operations on these sets are derived from the operations on $\mathbb{R}$.

Property (Inclusion)

For any two sets X and Y, set $X \diamond Y$ is defined so that

$$\forall x \in X, y \in Y,\ x \diamond y \in X \diamond Y.$$

By composition, $\forall x \in X, y \in Y, \ldots\ f(x, y, \ldots) \in F(X, Y, \ldots)$.

Definition (Interval Arithmetic)

For $u \in [\underline{u}, \overline{u}]$ and $v \in [\underline{v}, \overline{v}]$,

$$\begin{array}{rcl} u + v & \in & [\underline{u} + \underline{v}, \overline{u} + \overline{v}] \\ u - v & \in & [\underline{u} - \overline{v}, \overline{u} - \underline{v}] \\ & \vdots & \end{array}$$

Interval Arithmetic with Floating-Point Bounds

The inclusion property still holds if bounds are rounded outwards.

Definition (Interval Arithmetic with Floating-Point Bounds)

For $u \in U = [\underline{u}, \overline{u}]$ and $v \in V = [\underline{v}, \overline{v}]$,

$$\begin{array}{rcl} u + v & \in & [\bigtriangledown(\underline{u} + \underline{v}), \bigtriangleup(\overline{u} + \overline{v})] =: U + V \\ u - v & \in & [\bigtriangledown(\underline{u} - \overline{v}), \bigtriangleup(\overline{u} - \underline{v})] =: U - V \\ u \times v & \in & [\min(\bigtriangledown(\underline{u} \cdot \underline{v}), \bigtriangledown(\underline{u} \cdot \overline{v}), \bigtriangledown(\overline{u} \cdot \underline{v}), \bigtriangledown(\overline{u} \cdot \overline{v})), \max(\ldots)] \\ & \vdots & \end{array}$$

- Advantage: guaranteed arithmetic and constant-time operations.
- Drawback: correlation loss, $x - x \in X - X \neq [0, 0]$.

Tightening Intervals

How to reduce correlation when proving $\forall x \in X,\ f(x) \in I$?

1. Splitting intervals into subintervals:

$$f(x) \in \bigcup_i F(X_i) \quad \text{if } X \subseteq \bigcup_i X_i$$

2. Working at a higher order:

$$f(x) \in F(x_0) + (X - x_0) \times F'(X) \quad \text{if } x_0 \in X$$

3. Replacing intervals by models:

   Affine arithmetic; Taylor, Bernstein, Chebyshev models; etc.

4. ...

Kepler's Conjecture

Theorem (Hales, 1998)

Optimal density for packing 3D unit spheres is $\frac{\pi}{\sqrt{18}}$
(cubic close packing and hexagonal close packing).

Proof steps:

1. Enumerate all planar graphs.
2. Verify nonlinear inequalities.

Kepler's Conjecture

Example (I_751442360)

$\forall x \in X_{751442360}$,

$$\frac{\begin{array}{c}-x_1x_3 - x_2x_4 + x_1x_5 + x_3x_6 - x_5x_6 + \\ x_2(-x_2 + x_1 + x_3 - x_4 + x_5 + x_6)\end{array}}{\sqrt{4x_2\left(\begin{array}{c}x_2x_4(-x_2 + x_1 + x_3 - x_4 + x_5 + x_6) + \\ x_1x_5(x_2 - x_1 + x_3 + x_4 - x_5 + x_6) + \\ x_3x_6(x_2 + x_1 - x_3 + x_4 + x_5 - x_6) \\ -x_1x_3x_4 - x_2x_3x_5 - x_2x_1x_6 - x_4x_5x_6\end{array}\right)}} < \tan\left(\frac{\pi}{2} - \frac{\pi}{\sqrt{18}}\right)$$

1. Verified by a C program performing global optimization using floating-point interval arithmetic.
2. 12 referees during 4 years ⇒ "99% certain of the correctness".
3. Now being formalized: Flyspeck Project.