Primitive Floats in Coq

Guillaume Bertholon [1] Érik Martin-Dorel[2] Pierre Roux[3]

[1]École Normale Supérieure, Paris, France

[2]IRIT, Université Paul Sabatier, Toulouse, France

[3]ONERA, Toulouse, France

Tuesday 10 September 2019

ITP

Proofs involving floating-point computations (1/3)

Example (Square root)

- To prove that $a \in \mathbb{R}$ is non negative,
  we can exhibit r such that $a = r^2$ (typically $r = \sqrt{a}$).

Proofs involving floating-point computations (1/3)

Example (Square root)

- To prove that $a \in \mathbb{R}$ is non negative,
  we can exhibit r such that $a = r^2$ (typically $r = \sqrt{a}$).
- Using floating-point square root, $a \neq \mathrm{fl}(\sqrt{a})^2$

Proofs involving floating-point computations (1/3)

Example (Square root)

- To prove that $a \in \mathbb{R}$ is non negative,
  we can exhibit r such that $a = r^2$ (typically $r = \sqrt{a}$).
- Using floating-point square root, $a \neq \mathrm{fl}(\sqrt{a})^2$
- but one can subtract appropriate (tiny) c_a for which:
  if $\mathrm{fl}(\sqrt{a - c_a})$ succeeds then a is non negative

Proofs involving floating-point computations (2/3)

Example (Cholesky decomposition)

- To prove that a matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite we can similarly expose R such that $A = R^T R$ (since $x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$).

Proofs involving floating-point computations (2/3)

Example (Cholesky decomposition)

- To prove that a matrix $A \in \mathbb{R}^{n\times n}$ is positive semi-definite we can similarly expose R such that $A = R^T R$ (since $x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$).
- The Cholesky decomposition computes such a matrix R:

R **:=** 0;
for j **from** 1 **to** n **do**
  for i **from** 1 **to** $j-1$ **do**
    $R_{i,j}$ **:=** $\left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  od
  $R_{j,j}$ **:=** $\sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
od

Proofs involving floating-point computations (2/3)

Example (Cholesky decomposition)

- To prove that a matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite we can similarly expose R such that $A = R^T R$ (since $x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$).
- The Cholesky decomposition computes such a matrix R:

$R := 0$;
for j **from** 1 **to** n **do**
  for i **from** 1 **to** $j-1$ **do**
    $R_{i,j} := \left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  od
  $R_{j,j} := \sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
od

- With rounding errors $A \neq R^T R$
- but error is bounded and for some (tiny) $c_A \in \mathbb{R}$: if Cholesky succeeds on $A - c_A I$ then $A \succeq 0$.

Proofs involving floating-point computations (3/3)

Example (Interval Arithmetic)

- Datatype: interval = pair of (computable) real numbers
- E.g., $[3.1415, 3.1416] \ni \pi$
- Operations on intervals, e.g., $[2,4] - [0,1] := [2-1, 4-0] = [1,4]$, with the enclosure property: $\forall x \in [2,4],\ \forall y \in [0,1],\ x - y \in [1,4]$.
- Tool for bounding the range of functions

Proofs involving floating-point computations (3/3)

Example (Interval Arithmetic)

- Datatype: interval = pair of (computable) real numbers
- E.g., $[3.1415,\ 3.1416] \ni \pi$
- Operations on intervals, e.g., $[2,4] - [0,1] := [2-1, 4-0] = [1,4]$, with the enclosure property: $\forall x \in [2,4],\ \forall y \in [0,1],\ x - y \in [1,4]$.
- Tool for bounding the range of functions
- In practice, interval arithmetic can be efficiently implemented with floating-point arithmetic and directed roundings (towards $\pm\infty$).
- Thus floating-point computations (of interval bounds) can be used to prove numerical facts.

Motivations

- Coq offers some computation capabilities

$\leadsto$ which can be used in proofs

- Coq already offers efficient integers

Goal of this work

- Implement primitive computation in Coq with machine binary64 floats
- Instead of emulating floats with integers (about 1000x slower)

Agenda

Agenda

Coq, computation, and proof by reflection

Coq comes with a primitive notion of computation, called conversion.

Key feature of Coq's logic: the convertibility rule

In environment E, if $p : A$ and if A and B are convertible, then $p : B$.

So we can perform proofs by reflection:

Coq, computation, and proof by reflection

Coq comes with a primitive notion of computation, called conversion.

Key feature of Coq's logic: the convertibility rule

In environment E, if $p : A$ and if A and B are convertible, then $p : B$.

So we can perform proofs by reflection:

- Suppose that we want to prove G.
- We reify G and automatically prove that $f(c_1, \ldots) = \text{true} \Rightarrow G$,
  - by using a dedicated correctness lemma,
  - where f is a computable Boolean function.
  - So we only have to prove that $f(c_1, \ldots) = \text{true}$.
- We evaluate $f(c_1, \ldots)$.
- If the computation yields true:
  - This means that the type "$f(c_1, \ldots) = \text{true}$" is convertible with the type "$\text{true} = \text{true}$".
  - So we conclude by using reflexivity and the convertibility rule.

Computing with Coq in practice

Three main reduction tactics are available:

1984: `compute`: reduction machine

2004: `vm_compute`: virtual machine (byte-code)

2011: `native_compute`: compilation (native-code)

method	speed	TCB size
`compute`	+	+
`vm_compute`	++	++
`native_compute`	+++	+++

Efficient arithmetic in Coq

1994: `positive`, `N`, `Z` $\rightsquigarrow$ binary integers

2008: `bigN`, `bigZ`, `bigQ` $\rightsquigarrow$ binary trees of 31-bit machine integers

- Reference implementation in Coq (using lists of bits)
- Optimization with processor integers in `{vm,native}_compute`
- Implicit assumption that both implementations match

Efficient arithmetic in Coq

1994: `positive`, `N`, `Z` $\rightsquigarrow$ binary integers

2008: `bigN`, `bigZ`, `bigQ` $\rightsquigarrow$ binary trees of 31-bit machine integers

- Reference implementation in Coq (using lists of bits)
- Optimization with processor integers in `{vm,native}_compute`
- Implicit assumption that both implementations match

2019: `int` $\rightsquigarrow$ unsigned 63-bit machine integers + *primitive computation*

- Compact representation of integers in the kernel
- Efficient operations available for all reduction strategies
- Explicit axioms to specify the primitive operations

Floating-Point Values

Definition

A floating-point format $\mathbb{F}$ is a subset of $\mathbb{R}$. $x \in \mathbb{F}$ when

$$x = m\beta^e$$

for some m, $e \in \mathbb{Z}$, $|m| < \beta^p$ and $e_{\text{min}} \leq e \leq e_{\text{max}}$.

Floating-Point Values

Definition

A floating-point format $\mathbb{F}$ is a subset of $\mathbb{R}$. $x \in \mathbb{F}$ when

$$x = m\beta^e$$

for some m, $e \in \mathbb{Z}$, $|m| < \beta^p$ and $e_{\text{min}} \leq e \leq e_{\text{max}}$.

- m: *mantissa* of x
- β: *radix* of $\mathbb{F}$ (2 in practice)
- p: *precision* of $\mathbb{F}$
- e: *exponent* of x
- e_{min}: minimal exponent of $\mathbb{F}$
- e_{max}: maximal exponent of $\mathbb{F}$

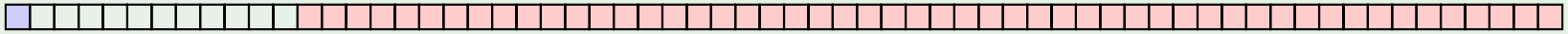
IEEE 754 standard

The IEEE 754 standard defines floating-point formats and operations.

Example

For binary64 format (type `double` in C): $\beta = 2$, $p = 53$ and $e_{min} = -1074$.

Binary representation:



sign exponent (11 bits) mantissa (52 bits)

$+$ Special values: $\pm\infty$ and NaNs (Not A Number, e.g., $0/0$ or $\sqrt{-1}$)

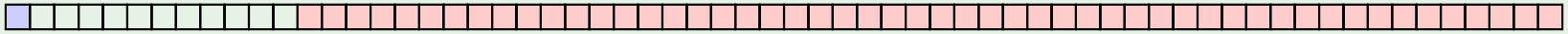
IEEE 754 standard

The IEEE 754 standard defines floating-point formats and operations.

Example

For binary64 format (type `double` in C): $\beta = 2$, $p = 53$ and $e_{min} = -1074$.

Binary representation:



sign   exponent (11 bits)   mantissa (52 bits)

+ Special values: $\pm\infty$ and NaNs (Not A Number, e.g., $0/0$ or $\sqrt{-1}$)

Remarks

- two zeros: $+0$ and -0 ($1/+0 = +\infty$ whereas $1/-0 = -\infty$)

IEEE 754 standard

The IEEE 754 standard defines floating-point formats and operations.

Example

For binary64 format (type `double` in C): $\beta = 2$, $p = 53$ and $e_{min} = -1074$.

Binary representation:

sign exponent (11 bits) mantissa (52 bits)

+ Special values: $\pm\infty$ and NaNs (Not A Number, e.g., $0/0$ or $\sqrt{-1}$)

Remarks

- two zeros: $+0$ and -0 ($1/+0 = +\infty$ whereas $1/-0 = -\infty$)
- many NaNs (used to carry error messages)

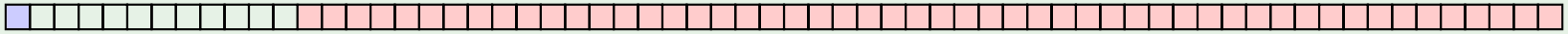
IEEE 754 standard

The IEEE 754 standard defines floating-point formats and operations.

Example

For binary64 format (type `double` in C): $\beta = 2$, $p = 53$ and $e_{min} = -1074$.

Binary representation:



sign exponent (11 bits) mantissa (52 bits)

+ Special values: $\pm\infty$ and NaNs (Not A Number, e.g., $0/0$ or $\sqrt{-1}$)

Remarks

- two zeros: $+0$ and -0 ($1/+0 = +\infty$ whereas $1/-0 = -\infty$)
- many NaNs (used to carry error messages)
- $+0 = -0$ but NaN $\neq$ NaN (for all NaN)

Flocq

Flocq is a Coq library formalizing floating-point arithmetic

- very *generic* formalization (multi-radix, multi-precision)
- linked with *real numbers* of the Coq standard library
- multiple models available
  - without overflow nor underflow
  - with underflow (either gradual or abrupt)
  - IEEE 754 binary format (used in Compcert)
- many classical results about roundings and specialized algorithms
- effective numerical computations

It is mainly developed by Sylvie Boldo and Guillaume Melquiond and available at http://flocq.gforge.inria.fr/

CoqInterval

CoqInterval is a Coq library formalizing interval arithmetic

- modular formalization involving Coq signatures and modules
- intervals with floating-point bounds
- radix-2 floating-point numbers (pairs of `bigZ`, no underflow/overflow)

⇝ *efficient* numerical computations

- support of elementary functions such as `exp`, `ln` and `atan`...
- tactics (`interval`, `interval_intro`) to automatically prove inequalities on real-valued expressions.

It is mainly developed by Guillaume Melquiond
and available at `http://coq-interval.gforge.inria.fr/`

Agenda

Workflow

1. Define a minimal working interface for the IEEE 754 binary64 format.
2. Define a fully-specified spec w.r.t. a minimal excerpt of Flocq.
3. Prepare a compatibility layer that could later be added to Flocq.
4. Implementation for `compute`, `vm_compute` and `native_compute`, at the OCaml and C levels.
5. Run some benchmarks.

Interface (1/4)

```
Require Import Floats.

(* contains *)

Parameter float : Set.
Parameter opp : float -> float.
Parameter abs : float -> float.

Variant float_comparison : Set :=
  | FEq | FLt | FGt | FNotComparable.
Variant float_class : Set :=
  | PNormal | NNormal | PSubn | NSubn | PZero | NZero
  | PInf | NInf | NaN.
Parameter compare : float -> float -> float_comparison.
Parameter classify : float -> float_class.
```

Interface (2/4)

```
Parameters mul add sub div : float -> float -> float.
Parameter sqrt : float -> float.
(* The value is rounded if necessary. *)
Parameter of_int63 : Int63.int -> float.
(* If input inside [0.5; 1.) then return its mantissa. *)
Parameter normfr_mantissa : float -> Int63.int.
Definition shift := (2101)%int63.  (* = 2*emax + prec *)
(* frshiftexp f = (m, e)
   s.t. m \in [0.5, 1) and f = m * 2^(e-shift) *)
Parameter frshiftexp : float -> float * Int63.int.
(* ldshiftexp f e = f * 2^(e-shift) *)
Parameter ldshiftexp : float -> Int63.int -> float.
Parameter next_up : float -> float.
Parameter next_down : float -> float.
```

Interface (3/4)

Computes but useless for proofs, we need a specification

```
Variant spec_float :=
  | S754_zero (s : bool)
  | S754_infinity (s : bool)
  | S754_nan
  | S754_finite (s : bool) (m : positive) (e : Z).

Definition SFopp x :=
  match x with
  | S754_zero sx => S754_zero (negb sx)
  | S754_infinity sx => S754_infinity (negb sx)
  | S754_nan => S754_nan
  | S754_finite sx mx ex => S754_finite (negb sx) mx ex
  end.
(* ... (mostly borrowed from Flocq) *)
```

Interface (4/4)

And axioms to link everything

```
Definition Prim2SF : float -> spec_float.
Definition SF2Prim : spec_float -> float.

Axiom opp_spec :
  forall x, Prim2SF (-x)%float = SFopp (Prim2SF x).
Axiom mul_spec :
  forall x y, Prim2SF (x * y)%float
    = SF64mul (Prim2SF x) (Prim2SF y).
(* ... *)
```

Not yet implemented:

- `roundToIntegral : mode -> float -> float`
- `convertToIntegral : mode -> float -> int`

Pitfalls

NaNs	their *payload* is hardware-dependent $\rightsquigarrow$ this could easily lead to a proof of `False`
Comparison	do not use IEEE 754 comparison for Leibniz equality (equates $+0$ and -0 whereas $\frac{1}{+0} = +\infty$ and $\frac{1}{-0} = -\infty$)
Primitive int63	are *unsigned* $\rightsquigarrow$ requires some care with signed exponents
OCaml floats	are *boxed* $\rightsquigarrow$ take care of garbage collector in `vm_compute` (and unboxed float arrays!)
x87 registers	$\rightsquigarrow$ double roundings (particularly with OCaml on 32 bits)

Pitfalls

NaNs: their *payload* is hardware-dependent
$\rightsquigarrow$ this could easily lead to a proof of `False`

Comparison: do not use IEEE 754 comparison for Leibniz equality
(equates $+0$ and -0 whereas $\frac{1}{+0} = +\infty$ and $\frac{1}{-0} = -\infty$)

Primitive int63: are *unsigned* $\rightsquigarrow$ requires some care with signed exponents

OCaml floats: are *boxed* $\rightsquigarrow$ take care of garbage collector in `vm_compute`
(and unboxed float arrays!)

x87 registers: $\rightsquigarrow$ double roundings (particularly with OCaml on 32 bits)

Parsing and pretty-printing

- easy solution: hexadecimal (e.g., `0xap-3`)
- ugly and unreadable for humans $\rightsquigarrow$ decimal (e.g., `1.25`)
- indeed, using 17 digits guarantees $parse \circ print$ to be the identity over binary64 (despite $parse$ not injective)
- decimal notations available in Coq 8.10

Agenda

Benchmarks (1/3)

[Demo]

- Measure the elapsed time with/without primitive floats for a reflexive proof tactic "`posdef_check`".

Source	Emulated floats	Primitive floats	Speedup
mat050	0.158s ±2.0%	0.008s ±0.0%	19.8x
mat100	1.162s ±1.3%	0.055s ±5.8%	21.1x
mat150	3.605s ±1.2%	0.176s ±2.2%	20.5x
mat200	8.684s ±0.2%	0.407s ±1.0%	21.3x
mat250	17.143s ±1.3%	0.801s ±0.3%	21.4x
mat300	30.005s ±1.2%	1.366s ±0.7%	22.0x
mat350	48.310s ±1.3%	2.146s ±0.1%	22.5x
mat400	70.193s ±1.4%	3.182s ±0.5%	22.1x

- We'd also like to measure the speed-up so obtained on the individual arithmetic operations!

Benchmarks (2/3) – vm_compute

Op	Source	Emulated floats CPU times (Op×2−Op)	Op time	Primitive floats CPU times (Op×1001−Op)	Op time	Speedup
add	mat200	10.783±0.9% − 8.381±2.8%	2.403s	15.718±0.5% − 0.446±1.1%	0.015s	157.3x
add	mat250	21.463±1.7% − 16.405±1.5%	5.058s	30.622±0.6% − 0.818±0.6%	0.030s	169.7x
add	mat300	37.430±1.4% − 28.630±1.4%	8.799s	53.122±2.4% − 1.400±0.5%	0.052s	170.1x
add	mat350	59.420±0.8% − 45.945±2.9%	13.475s	84.194±0.8% − 2.190±0.5%	0.082s	164.3x
add	mat400	87.783±0.9% − 66.173±1.7%	21.610s	127.562±8.5% − 3.214±0.3%	0.124s	173.8x
mul	mat200	12.212±1.4% − 8.381±2.8%	3.831s	16.096±3.0% − 0.446±1.1%	0.016s	244.8x
mul	mat250	24.517±1.4% − 16.405±1.5%	8.112s	31.118±3.7% − 0.818±0.6%	0.030s	267.7x
mul	mat300	42.844±1.7% − 28.630±1.4%	14.214s	53.249±0.8% − 1.400±0.5%	0.052s	274.1x
mul	mat350	68.228±1.5% − 45.945±2.9%	22.283s	84.332±0.7% − 2.190±0.5%	0.082s	271.3x
mul	mat400	99.722±1.5% − 66.173±1.7%	33.549s	125.742±0.8% − 3.214±0.3%	0.123s	273.8x

Table: Computation time for individual operations obtained by subtracting the CPU time of a normal execution from that of a modified execution where the specified operation is computed twice (resp. 1001 times). Each timing is measured 5 times. The table indicates the corresponding average and relative error among the 5 samples (using `vm_compute`).

Benchmarks (3/3) – native_compute

Op	Source	Emulated floats CPU times (Op×2−Op)	Op time	Primitive floats CPU times (Op×1001−Op)	Op time	Speedup
add	mat200	2.243±1.4% − 1.780±1.7%	0.463s	17.681±1.4% − 0.221±0.9%	0.017s	26.5x
add	mat250	4.486±4.2% − 3.411±3.1%	1.075s	34.290±0.7% − 0.368±1.5%	0.034s	31.7x
add	mat300	7.249±1.2% − 5.825±4.6%	1.424s	59.565±2.5% − 0.553±0.9%	0.059s	24.1x
add	mat350	11.664±3.8% − 9.275±3.5%	2.389s	93.818±1.1% − 0.816±0.8%	0.093s	25.7x
add	mat400	17.073±2.9% − 13.142±0.9%	3.930s	141.973±2.6% − 1.184±0.9%	0.141s	27.9x
mul	mat200	2.478±1.5% − 1.780±1.7%	0.698s	17.807±1.1% − 0.221±0.9%	0.018s	39.7x
mul	mat250	4.824±2.4% − 3.411±3.1%	1.412s	35.144±2.1% − 0.368±1.5%	0.035s	40.6x
mul	mat300	8.413±2.4% − 5.825±4.6%	2.588s	60.660±2.2% − 0.553±0.9%	0.060s	43.1x
mul	mat350	13.211±2.4% − 9.275±3.5%	3.937s	97.248±1.0% − 0.816±0.8%	0.096s	40.8x
mul	mat400	19.269±1.5% − 13.142±0.9%	6.127s	138.607±2.3% − 1.184±0.9%	0.137s	44.6x

Table: Computation time for individual operations obtained by subtracting the CPU time of a normal execution from that of a modified execution where the specified operation is computed twice (resp. 1001 times). Each timing is measured 5 times. The table indicates the corresponding average and relative error among the 5 samples (using `native_compute`).

Agenda

Concluding remarks

Wrap-up

- Implementing machine-efficient floats in Coq's low-level layers
- Focus on binary64 and on portability (IEEE 754, no NaN payloads. . .)
- Builds on the methodology of primitive integers ($\sim$2x / 31-bit retro.)
- Speedup of at least 150x for addition, 250x for multiplication

Concluding remarks

Wrap-up

- Implementing machine-efficient floats in Coq's low-level layers
- Focus on binary64 and on portability (IEEE 754, no NaN payloads...)
- Builds on the methodology of primitive integers (∼2x / 31-bit retro.)
- Speedup of at least 150x for addition, 250x for multiplication

Discussion and perspectives

- on-going pull request https://github.com/coq/coq/pull/9867
- investigate if `next_{up,down}` could be emulated (and at which cost)
- nice applications (interval arithmetic with Coq.Interval, other ideas?)

Thank you!

Questions

?

https://github.com/coq/coq/pull/9867