Primitive Floats in Coq

Guillaume Bertholon\textsuperscript{1} \quad Érik Martin-Dorel\textsuperscript{2} \quad Pierre Roux\textsuperscript{3}

\begin{align*}
\textsuperscript{1} & \text{École Normale Supérieure, Paris, France} \\
\textsuperscript{2} & \text{IRIT, Université Paul Sabatier, Toulouse, France} \\
\textsuperscript{3} & \text{ONERA, Toulouse, France}
\end{align*}

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, $a \neq \text{fl}(\sqrt{a})^2$.
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, $a \neq \text{fl}(\sqrt{a})^2$
- but one can subtract appropriate (tiny) $c_a$ for which: if $\text{fl}(\sqrt{a - c_a})$ succeeds then $a$ is non negative
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 $x^T \left( R^T R \right) x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$).
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 $x^T (R^T R) x = (R x)^T (R x) = \|R x\|_2^2 \geq 0$).
- The Cholesky decomposition computes such a matrix $R$:

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

\[
\begin{align*}
R & := 0; \\
\text{for } j \text{ from } 1 \text{ to } n \text{ do} \\
\quad & \text{for } i \text{ from } 1 \text{ to } j - 1 \text{ do} \\
\quad & \quad R_{i,j} := \left( A_{i,j} - \sum_{k=1}^{i-1} R_{k,i} R_{k,j} \right) / R_{i,i}; \\
\quad & \text{od} \\
\quad R_{j,j} & := \sqrt{M_{j,j} - \sum_{k=1}^{j-1} R_{k,j}^2}; \\
\quad & \text{od}
\end{align*}
\]

- 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 \( A \succeq 0 \).
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
  → 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

1. Introduction
2. State of the art
3. Implementation
4. Numerical results
5. Conclusion
1 Introduction

2 State of the art

3 Implementation

4 Numerical results

5 Conclusion
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)

<table>
<thead>
<tr>
<th>method</th>
<th>speed</th>
<th>TCB size</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>vm_compute</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>native_compute</td>
<td>+++</td>
<td>+++</td>
</tr>
</tbody>
</table>
Efficient arithmetic in Coq

1994: positive, \( \mathbb{N}, \mathbb{Z} \) \( \hookrightarrow \) binary integers

2008: big\( \mathbb{N} \), big\( \mathbb{Z} \), big\( \mathbb{Q} \) \( \hookrightarrow \) binary trees of 31-bit machine integers
   - Reference implementation in Coq (using lists of bits)
   - Optimization with processor integers in \( \{\text{vm}, \text{native}\}_\text{compute} \)
   - Implicit assumption that both implementations match
Efficient arithmetic in Coq

1994: positive, N, Z \rightarrow \text{binary integers}

2008: bigN, bigZ, bigQ \rightarrow \text{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 \rightarrow \text{unsigned 63-bit machine integers} + \text{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 $F$ is a subset of $\mathbb{R}$. $x \in F$ 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 $x$
- $\beta$: radix of $F$ (2 in practice)
- $p$: precision of $F$
- $e$: exponent of $x$
- $e_{\text{min}}$: minimal exponent of $F$
- $e_{\text{max}}$: maximal exponent of $F$
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:

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent (11 bits)</th>
<th>mantissa (52 bits)</th>
</tr>
</thead>
</table>

+ 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 $e_{\text{min}} = -1074$.

Binary representation:

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent (11 bits)</th>
<th>mantissa (52 bits)</th>
</tr>
</thead>
</table>

+ 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 $e_{\text{min}} = -1074$.

Binary representation:

- Sign: 1 bit
- 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 $e_{\text{min}} = -1074$.

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)

\( \Rightarrow \) efficient numerical computations

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

It is mainly developed by Guillaume Melquiond
and available at\footnote{http://coq-interval.gforge.inria.fr/}
Agenda

1. Introduction
2. State of the art
3. Implementation
4. Numerical results
5. Conclusion
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.
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) *)
And axioms to link everything

**Definition** Prim2SF : float → spec_float.
**Definition** SF2Prim : spec_float → float.

**Axiom** opp_spec :
\[ \text{forall } x, \text{Prim2SF} (-x)\%\text{float} = \text{SFopp} (\text{Prim2SF} x). \]

**Axiom** mul_spec :
\[ \text{forall } x \ y, \text{Prim2SF} (x * y)\%\text{float} = \text{SF64mul} (\text{Prim2SF} x) (\text{Prim2SF} y). \]

(* ... * )

Not yet implemented:

- roundToIntegral : mode → float → float
- convertToIntegral : mode → float → int
Pitfalls

**NaNs** their *payload* is hardware-dependent
\[ \leadsto \text{this could easily lead to a proof of } \texttt{False} \]

**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* \( \leadsto \text{requires some care with signed exponents} \)

**OCaml floats** are *boxed* \( \leadsto \text{take care of garbage collector in } \texttt{vm\_compute} \)
(and unboxed float arrays!)

**x87 registers** \( \leadsto \text{double roundings (particularly with OCaml on 32 bits)} \)
Pitfalls

**NaNs** their *payload* is hardware-dependent

\[ \sim \Rightarrow \text{this could easily lead to a proof of } \text{False} \]

**Comparison** do not use IEEE 754 comparison for Leibniz equality

(\text{equates } +0 \text{ and } -0 \text{ whereas } \frac{1}{+0} = +\infty \text{ and } \frac{1}{-0} = -\infty)

**Primitive int63** are *unsigned* \( \sim \Rightarrow \text{requires some care with signed exponents} \)

**OCaml floats** are *boxed* \( \sim \Rightarrow \text{take care of garbage collector in } \text{vm\_compute} \)

(and unboxed float arrays!)

**x87 registers** \( \sim \Rightarrow \text{double roundings (particularly with OCaml on 32 bits)} \)

**Parsing and pretty-printing**

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

1. Introduction
2. State of the art
3. Implementation
4. Numerical results
5. Conclusion
[Demo]

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

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

- We’d also like to measure the speed-up so obtained on the individual arithmetic operations!
Benchmarks (2/3) – vm_compute

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Emulated floats</th>
<th>Primitive floats</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU times (Op × 2 − Op)</td>
<td>Op time</td>
<td>CPU times (Op × 1001 − Op)</td>
</tr>
<tr>
<td>add</td>
<td>mat200</td>
<td>10.783±0.9% − 8.381±2.8%</td>
<td>2.403s</td>
<td>15.718±0.5% − 0.446±1.1%</td>
</tr>
<tr>
<td>add</td>
<td>mat250</td>
<td>21.463±1.7% − 16.405±1.5%</td>
<td>5.058s</td>
<td>30.622±0.6% − 0.818±0.6%</td>
</tr>
<tr>
<td>add</td>
<td>mat300</td>
<td>37.430±1.4% − 28.630±1.4%</td>
<td>8.799s</td>
<td>53.122±2.4% − 1.400±0.5%</td>
</tr>
<tr>
<td>add</td>
<td>mat350</td>
<td>59.420±0.8% − 45.945±2.9%</td>
<td>13.475s</td>
<td>84.194±0.8% − 2.190±0.5%</td>
</tr>
<tr>
<td>add</td>
<td>mat400</td>
<td>87.783±0.9% − 66.173±1.7%</td>
<td>21.610s</td>
<td>127.562±8.5% − 3.214±0.3%</td>
</tr>
<tr>
<td>mul</td>
<td>mat200</td>
<td>12.212±1.4% − 8.381±2.8%</td>
<td>3.831s</td>
<td>16.096±3.0% − 0.446±1.1%</td>
</tr>
<tr>
<td>mul</td>
<td>mat250</td>
<td>24.517±1.4% − 16.405±1.5%</td>
<td>8.112s</td>
<td>31.118±3.7% − 0.818±0.6%</td>
</tr>
<tr>
<td>mul</td>
<td>mat300</td>
<td>42.844±1.7% − 28.630±1.4%</td>
<td>14.214s</td>
<td>53.249±0.8% − 1.400±0.5%</td>
</tr>
<tr>
<td>mul</td>
<td>mat350</td>
<td>68.228±1.5% − 45.945±2.9%</td>
<td>22.283s</td>
<td>84.332±0.7% − 2.190±0.5%</td>
</tr>
<tr>
<td>mul</td>
<td>mat400</td>
<td>99.722±1.5% − 66.173±1.7%</td>
<td>33.549s</td>
<td>125.742±0.8% − 3.214±0.3%</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Emulated floats</th>
<th>Primitive floats</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU times (Op×2−Op)</td>
<td>Op time</td>
<td>CPU times (Op×1001−Op)</td>
</tr>
<tr>
<td>add</td>
<td>mat200</td>
<td>2.243±1.4% − 1.780±1.7%</td>
<td>0.463s</td>
<td>17.681±1.4% − 0.221±0.9%</td>
</tr>
<tr>
<td>add</td>
<td>mat250</td>
<td>4.486±4.2% − 3.411±3.1%</td>
<td>1.075s</td>
<td>34.290±0.7% − 0.368±1.5%</td>
</tr>
<tr>
<td>add</td>
<td>mat300</td>
<td>7.249±1.2% − 5.825±4.6%</td>
<td>1.424s</td>
<td>59.565±2.5% − 0.553±0.9%</td>
</tr>
<tr>
<td>add</td>
<td>mat350</td>
<td>11.664±3.8% − 9.275±3.5%</td>
<td>2.389s</td>
<td>93.818±1.1% − 0.816±0.8%</td>
</tr>
<tr>
<td>add</td>
<td>mat400</td>
<td>17.073±2.9% − 13.142±0.9%</td>
<td>3.930s</td>
<td>141.973±2.6% − 1.184±0.9%</td>
</tr>
<tr>
<td>mul</td>
<td>mat200</td>
<td>2.478±1.5% − 1.780±1.7%</td>
<td>0.698s</td>
<td>17.807±1.1% − 0.221±0.9%</td>
</tr>
<tr>
<td>mul</td>
<td>mat250</td>
<td>4.824±2.4% − 3.411±3.1%</td>
<td>1.412s</td>
<td>35.144±2.1% − 0.368±1.5%</td>
</tr>
<tr>
<td>mul</td>
<td>mat300</td>
<td>8.413±2.4% − 5.825±4.6%</td>
<td>2.588s</td>
<td>60.660±2.2% − 0.553±0.9%</td>
</tr>
<tr>
<td>mul</td>
<td>mat350</td>
<td>13.211±2.4% − 9.275±3.5%</td>
<td>3.937s</td>
<td>97.248±1.0% − 0.816±0.8%</td>
</tr>
<tr>
<td>mul</td>
<td>mat400</td>
<td>19.269±1.5% − 13.142±0.9%</td>
<td>6.127s</td>
<td>138.607±2.3% − 1.184±0.9%</td>
</tr>
</tbody>
</table>

**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

1. Introduction
2. State of the art
3. Implementation
4. Numerical results
5. Conclusion
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
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