Primitive Floats in Coq

Érik Martin-Dorel\textsuperscript{1} Pierre Roux\textsuperscript{2}
with a lot of work from Guillaume Bertholon

\textsuperscript{1}IRIT, Université Paul Sabatier, Toulouse, France

\textsuperscript{2}ONERA, Toulouse, France

Thursday 20 June 2019

FEANICSES Workshop
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
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 $x^T \left( R^T R \right) x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$).

---

É. Martin-Dorel, P. Roux

Primitive Floats in Coq
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 $R$:

```plaintext
R := 0;
for j from 1 to n do
    for i from 1 to j - 1 do
        Ri,j := (Ai,j - Σk=1i-1 Rk,i Rk,j) / Ri,i;
    od
    Rj,j := \sqrt{Mj,j - Σk=1j-1 Rk,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 $x^T (R^T R) x = (R x)^T (R x) = \|R x\|_2^2 \geq 0$).
- The Cholesky decomposition computes such a matrix $R$:

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

- 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 $A \succeq 0$. 

É. Martin-Dorel, P. Roux

Primitive Floats in Coq
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
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
Agenda

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 $p : B$. 
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 $G$. 
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 $f(c_1, \ldots) = \text{true} \Rightarrow G$,
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,
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.
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 $f(c_1, \ldots) = \text{true}$.
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 $f(c_1, \ldots)$.  
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:
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 “true = true”.
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>

É. Martin-Dorel, P. Roux

Primitive Floats in Coq
Efficient arithmetic in Coq

1994: positive, $\mathbb{N}$, $\mathbb{Z}$ $\mapsto$ binary integers

2008: $\text{bigN}$, $\text{bigZ}$, $\text{bigQ}$ $\mapsto$ binary trees of 31-bit machine integers
- Reference implementation in Coq (using lists of bits)
- Optimization with processor integers in $\{\text{vm, native}\}_\text{compute}$
- Implicit assumption that both implementations match
1994: positive, \( \mathbb{N}, \mathbb{Z} \mapsto \) binary integers

2008: \( \text{bigN, bigZ, bigQ} \mapsto \) 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: \( \text{int} \mapsto \) unsigned 63-bit machine integers + \textit{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 $x$
- $\beta$: radix of $\mathbb{F}$ (2 in practice)
- $p$: precision of $\mathbb{F}$
- $e$: exponent of $x$
- $e_{\text{min}}$: minimal exponent of $\mathbb{F}$
- $e_{\text{max}}$: 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 $e_{\text{min}} = -1074$.

Binary representation:

```
+ 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:

```
+ 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:

- 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, \text{no underflow/overflow})

\text{efficient} numerical computations

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

It is mainly developed by Guillaume Melquiond
and available at \url{http://coq-interval.gforge.inria.fr/}
<table>
<thead>
<tr>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>2. State of the art</td>
</tr>
<tr>
<td>3. Implementation</td>
</tr>
<tr>
<td>4. Numerical results</td>
</tr>
<tr>
<td>5. Conclusion</td>
</tr>
</tbody>
</table>
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 \texttt{compute}, \texttt{vm\_compute} and \texttt{native\_compute}, at the OCaml and C levels.
5. Run some benchmarks.
## Interface (1/4)

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

É. Martin-Dorel, P. Roux

Primitif Flots in Coq
Interface (3/4)

Computes but useless for proofs, we need a specification

**Inductive** `spec_float` :=
- `S754_zero : bool → spec_float`
- `S754_infinity : bool → spec_float`
- `S754_nan : spec_float`
- `S754_finite : bool → positive → Z → spec_float`.

**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 $\rightarrow$ spec_float.

**Definition** SF2Prim : spec_float $\rightarrow$ float.

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

**Axiom** FPmult_SFmult :
\[ \forall x \ y, \text{Prim2SF} (x * y)\%\text{float} = \text{SF64mult} (\text{Prim2SF} x) (\text{Prim2SF} y). \]

(* ... *)

Not yet implemented:
- **roundToIntegral** : mode $\rightarrow$ float $\rightarrow$ float
- **convertToIntegral** : mode $\rightarrow$ float $\rightarrow$ int
Pitfalls

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

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

**OCaml floats** are *boxed* \( \mapsto \text{take care of garbage collector in } \text{vm}_{\text{compute}} \)
(and unboxed float arrays!)

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

NaNs their *payload* is hardware-dependent
\[ \leadsto \] 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* \[ \leadsto \] requires some care with signed exponents

OCaml floats are *boxed* \[ \leadsto \] take care of garbage collector in *vm_compute*
(and unboxed float arrays!)

x87 registers \[ \leadsto \] double roundings (particularly with OCaml on 32 bits)

Parsing and pretty-printing
- easy solution: hexadecimal (e.g., \( 0xap-3 \))
- ugly and unreadable for humans \[ \leadsto \] 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
Benchmarks (1/3)

[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!
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 \texttt{vm\_compute}).
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