Hardware floating-point computations in Coq

Guillaume Bertholon[1] Érik Martin-Dorel[2] Guillaume Melquiond[3]
Pierre Roux[4]

[1](2019) École Normale Supérieure, Paris, France

[2] IRIT, Université Toulouse III – Paul Sabatier, Toulouse, France
https://linktr.ee/erikmd

[3] Inria, Université Paris-Saclay, Gif-sur-Yvette, France

[4] ONERA, Toulouse, France

May 26th, 2023

Certified and Symbolic-Numeric Computation, Lyon

Computing within Coq: Example

```
From mathcomp Require Import all_ssreflect.
Eval compute in filter prime (mkseq (fun n ⇒ n) 100).

⇝ [:: 2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41;
      43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]
   : seq nat
```

Datatypes for arithmetic in Coq

1984: birth of Coq

1989: primitive inductive definitions, e.g. `nat` $\rightsquigarrow$ radix-1 integers

Datatypes for efficient arithmetic in Coq

1984: birth of Coq

1989: primitive inductive definitions, e.g. `nat` $\rightsquigarrow$ radix-1 integers

1994: `positive`, `N`, `Z` $\rightsquigarrow$ radix-2 integers

Datatypes for efficient arithmetic in Coq

1984: birth of Coq

1989: primitive inductive definitions, e.g. `nat` $\rightsquigarrow$ radix-1 integers

1994: `positive`, `N`, `Z` $\rightsquigarrow$ radix-2 integers

2006: `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

Datatypes for efficient arithmetic in Coq

1984: birth of Coq

1989: primitive inductive definitions, e.g. `nat` $\rightsquigarrow$ radix-1 integers

1994: `positive`, `N`, `Z` $\rightsquigarrow$ radix-2 integers

2006: `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

2013: `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

Datatypes for efficient arithmetic in Coq

1984: birth of Coq

1989: primitive inductive definitions, e.g. `nat` $\rightsquigarrow$ radix-1 integers

1994: `positive`, `N`, `Z` $\rightsquigarrow$ radix-2 integers

2006: `bigN`, `bigZ`, `bigQ` $\rightsquigarrow$ binary trees of 31-bit machine integers

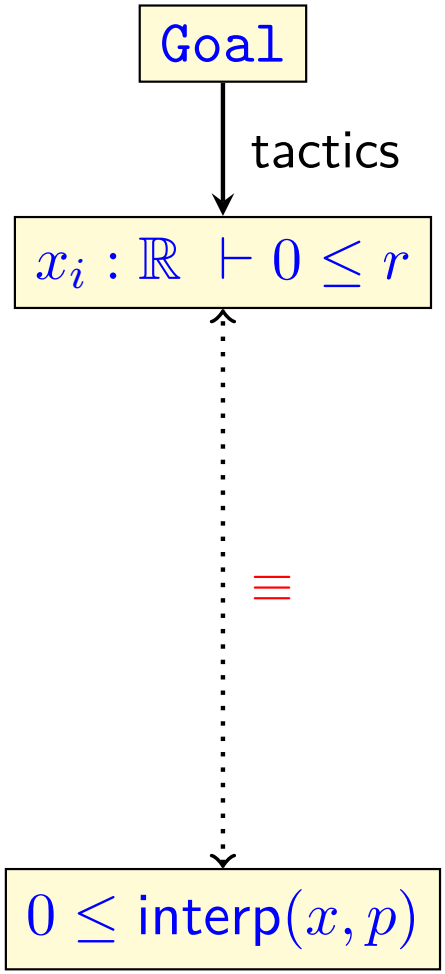
- Reference implementation in Coq (using lists of bits)
- Optimization with processor integers in `{vm,native}_compute`
- Implicit assumption that both implementations match

2013: `int` $\rightsquigarrow$ unsigned 63-bit machine integers + "primitive computation"

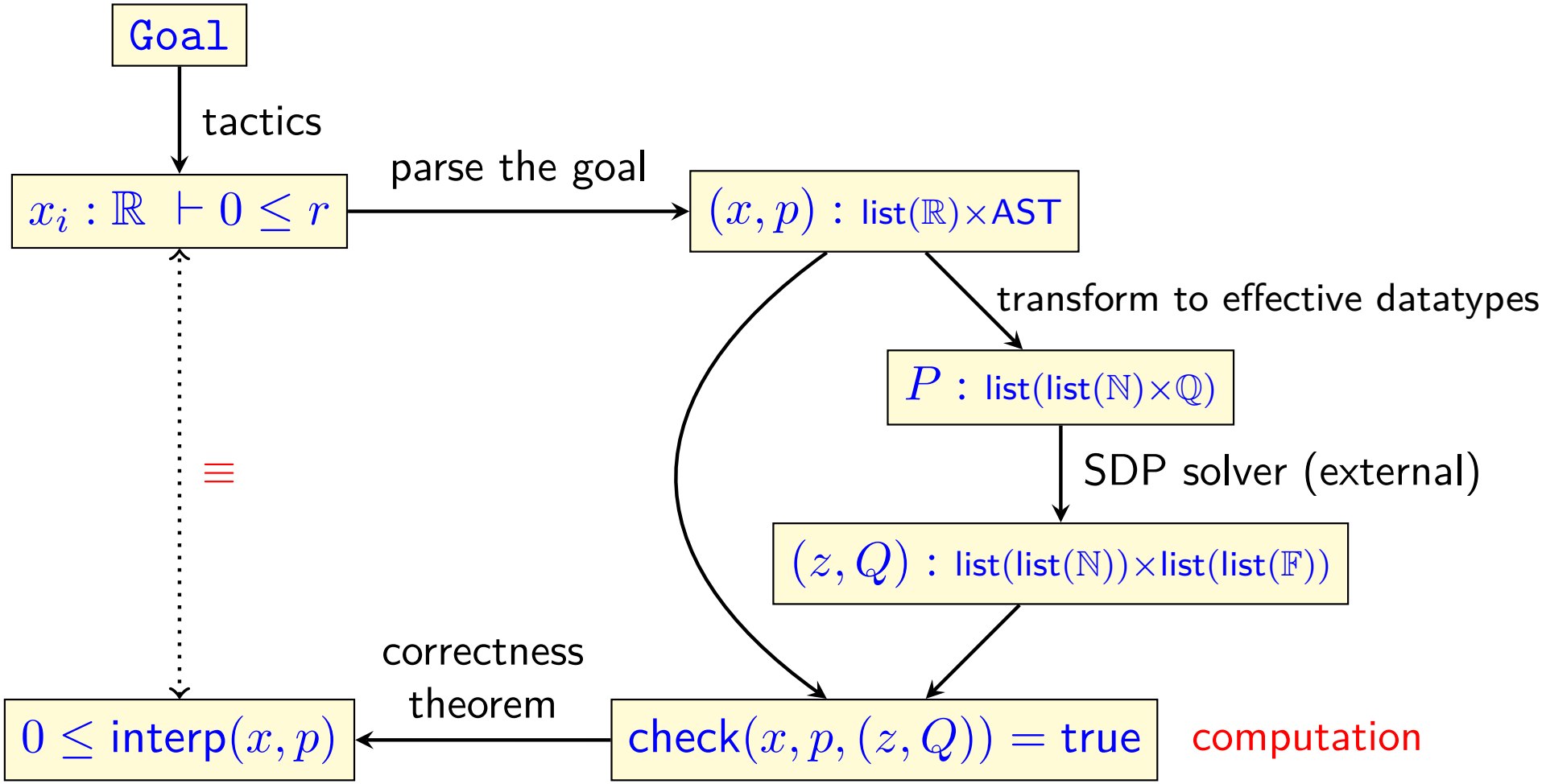
- Compact representation of integers in the kernel
- Efficient operations available for all reduction strategies
- Explicit axioms to specify the primitive operations

2019: `float` $\rightsquigarrow$ binary64 machine floating-point numbers (this work).

The validsdp tactic: relying on proof by reflection in Coq



The validsdp tactic: relying on proof by reflection in Coq



Wrap-up and Positioning

- Coq offers some computation capabilities
  and computations can be used in proofs
- Coq offers efficient integers (63-bit or multiple-precision)
- Until now, floating-point numbers were emulated with integers
  $\rightarrow$ Bottleneck in ValidSDP and CoqInterval libraries

Wrap-up and Positioning

- Coq offers some computation capabilities
  and computations can be used in proofs
- Coq offers efficient integers (63-bit or multiple-precision)
- Until now, floating-point numbers were emulated with integers
  $\rightarrow$ Bottleneck in ValidSDP and CoqInterval libraries
- Goal 1: Implement primitive floats (binary64) in Coq's standard library
- Goal 2: Refactor ValidSDP and CoqInterval to use primitive floats
- Goal 3: Evaluate the performance gain

Agenda

Floating-point formats

Definition

A floating-point format $\mathbb{F}$ is a subset of $\mathbb{R}$. $x \in \mathbb{F}$ iff $x \in \{\pm\infty, \mathsf{NaN}\}$ or

$$x = m\beta^e$$

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

- 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 $\mathbb{F}$

red: parameters of the format

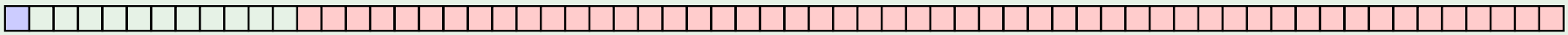
IEEE 754 standard

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

Example

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

Binary representation:



sign exponent (11 bits) mantissa (52 bits)

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

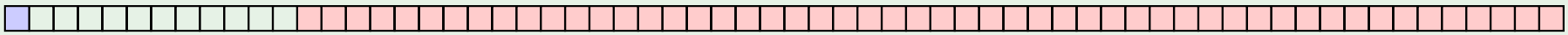
IEEE 754 standard

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:



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 $1/-0 = -\infty$)

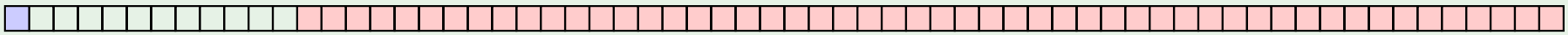
IEEE 754 standard

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

Example

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

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 NaN values (used to carry error messages)

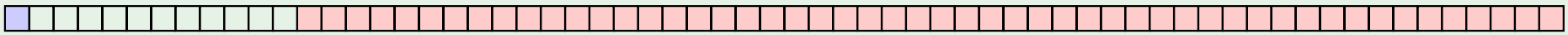
IEEE 754 standard

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 NaN values (used to carry error messages)
- $+0 = -0$ but NaN $\neq$ NaN (for all NaN)

Overview of the Flocq library [Boldo, Melquiond, 2011]

(see also Guillaume Melquiond's talk)

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

https://flocq.gitlabpages.inria.fr/

Applied workflow

1. Define a minimal interface for the IEEE 754 binary64 format.
2. Define a fully-specified spec reusing a minimal excerpt of Flocq.
3. Setup a compatibility layer and add it to Flocq.
4. Implement support for all reduction tactics, in OCaml and C layers.
5. Add convenience features (warnings, decimal/hexadecimal notations...)

Interface and Specification (1/4)

```
From Coq 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 and Specification (2/4)

```
Parameters mul add sub div : float → float → float.
Parameter sqrt : float → float.
(* using correct rounding with the dflt rounding mode. *)
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 ∈ [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 and Specification (3/4)

All this 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.
(* and so on (mostly borrowed from Flocq) *)
```

Interface and Specification (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).
(* and likewise for other operators. *)
```

Pitfalls

Specification issues Naturally, axioms are in the Trusted Computing Base.

Portability is critical The implementation of all reduction tactics must return the same results, whatever is the IEEE 754 compliant processor used.

NaN payloads are hardware-dependent (weakly IEEE-754 specified)
$\rightsquigarrow$ this could easily lead to a proof of `False`

x87 registers Double-roundings issues (especially with OCaml on 32 bits)

Comparisons Can't use IEEE 754 comparison for Coq's standard 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

Parsing and pretty-printing

- hexadecimal (`0xap-3`) & decimal (`1.25`) notions in `float_scope`
- formally proved that using 17 decimal digits entails $(\text{parse} \circ \text{print})$ is the identity over binary64

Bugs identified and fixed since 2019

Specification issues:

#12483 Incorrect spec of `leb` (had written `Le`, i/o `Lt|Eq`)

#16096 Incorrect spec of `classify` (the value of the sign bit was reversed)

Interaction with other primitive types:

#15070 Primitive persistent arrays (dev. concurrently with primitive floats) led to an incompatibility w.r.t. OCaml runtime memory invariants.

Bugs identified and fixed since 2019

Specification issues:

#12483 Incorrect spec of `leb` (had written `Le`, i/o `Lt|Eq`) → new warning.

#16096 Incorrect spec of `classify` (the value of the sign bit was reversed)

Interaction with other primitive types:

#15070 Primitive persistent arrays (dev. concurrently with primitive floats) led to an incompatibility w.r.t. OCaml runtime memory invariants.

Agenda

Overview of the ValidSDP library [Martin-Dorel, Roux, 2017]

(see also Pierre Roux's talk)

ValidSDP is a package of Coq tactics for multivariate polynomial positivity

- provide tactics: `validsdp`, `validsdp_intro` (and `posdef_check`)
- input: polynomials inequalities under polynomial constraints involving real-valued variables and rational constants
- use proof-by-reflection and `Ltac2` meta-programming
- use off-the-shelf SDP solvers as untrusted oracles
- use symbolic-numeric computations (matrices and floating-point arithmetic)

https://github.com/validsdp/validsdp#readme

Refactoring ValidSDP to add support for primitive floats

- initially: several `Record`s formalizing floating-point formats based on "the standard model of FP arith." (involving ε and η error bounds)

Refactoring ValidSDP to add support for primitive floats

- initially: several `Record`s formalizing floating-point formats based on "the standard model of FP arith." (involving ε and η error bounds)

- outcome: fortunately, the already formalized structures were precise enough to account for underflows, overflows, and NaNs;

  we just needed to implement two instances (binary64, $\circ :=$ RNE) and (binary64, $\circ :=$ RU); without needing to alter the existing `Record`s.

Overview of the CoqInterval library [Melquiond, 2008]

(see also Guillaume Melquiond's talk)

CoqInterval is a Coq library formalizing Interval Arithmetic

- tactics to automatically prove inequalities on real-valued expressions (`interval`, `interval_intro`, `integral`, `integral_intro`, etc.)
- modular formalization involving Coq signatures and modules
- intervals with floating-point bounds
- radix-2 floating-point numbers (pairs of `bigZ`, no underflow/overflow)

$\rightsquigarrow$ *efficient* numerical computations

- support of elementary functions such as `exp`, `ln` and `atan`

https://coqinterval.gitlabpages.inria.fr/

Refactoring CoqInterval to add support for primitive floats

- initially: the basic operators are exactly-rounded, using emulated, directed roundings. Example spec:

```
Parameter add_correct : forall mode p x y,
  toX (add mode p x y)
  = Xround radix mode (prec p) (Xadd (toX x) (toX y)).
```

Refactoring CoqInterval to add support for primitive floats

- initially: the basic operators are exactly-rounded, using emulated, directed roundings. Example spec:

```
Parameter add_correct : forall mode p x y,
  toX (add mode p x y)
  = Xround radix mode (prec p) (Xadd (toX x) (toX y)).
```

- use hardware rounding-to-nearest along with `next_up`, `next_down`
- we obtain a new interval arithmetic kernel without exact rounding ($\rightsquigarrow$ trade off exact-computation proofs for performance). Spec:

```
Parameter add_UP_correct : forall p x y,
  x ≠ −∞ → y ≠ −∞ → (add_UP p x y) ≠ −∞ ∧
  le_upper (Xadd (toX x) (toX y)) (toX (add_UP p x y)).
```

Refactoring CoqInterval to add support for primitive floats

- initially: the basic operators are exactly-rounded, using emulated, directed roundings. Example spec:

```
Parameter add_correct : forall mode p x y,
  toX (add mode p x y)
  = Xround radix mode (prec p) (Xadd (toX x) (toX y)).
```

- use hardware rounding-to-nearest along with `next_up`, `next_down`
- we obtain a new interval arithmetic kernel without exact rounding ($\rightsquigarrow$ trade off exact-computation proofs for performance). Spec:

```
Parameter add_UP_correct : forall p x y,
  x ≠ −∞ → y ≠ −∞ → (add_UP p x y) ≠ −∞ ∧
  le_upper (Xadd (toX x) (toX y)) (toX (add_UP p x y)).
```

- Other adaptations needed: some functions were incompatible with overflows, or just with the fixed precision of hardware floating-point numbers. Overall, the refactoring led to (+9700, −4300) LOC.

Agenda

Computing within Coq: Several powerful tactics

Three main reduction tactics are available:

1984: `compute`: reduction machine

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

2011: `native_compute`: compilation (ocaml native-code; dynlink)

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

Computing within Coq: Several powerful tactics

Three main reduction tactics are available:

1984: `compute`: reduction machine

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

2011: `native_compute`: compilation (ocaml native-code; dynlink)

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

Trusted Computing Base

Benchmarks (1/6) – ValidSDP

- ValidSDP: Measure the elapsed time with/without primitive floats for the tactic "`posdef_check`" using `vm_compute`.

Source	Emulated	Hardware	Speedup
mat0050	0.228s ±9.5%	0.013s ±16.7%	17.3×
mat0100	1.451s ±2.4%	0.113s ±8.2%	12.9×
mat0150	4.572s ±6.8%	0.276s ±13.0%	16.6×
mat0200	10.724s ±3.4%	0.557s ±9.8%	19.2×
mat0250	21.839s ±1.2%	1.032s ±3.5%	21.2×
mat0300	37.706s ±1.6%	1.810s ±4.7%	20.8×
mat0350	60.616s ±1.5%	2.802s ±4.1%	21.6×
mat0400	89.343s ±1.5%	4.110s ±0.9%	21.7×

Benchmarks (2/6) – ValidSDP

- ValidSDP: measure the speed-up on the individual arithmetic operations (for a Cholesky decomposition of random square matrices)

Op	compute	Emulated CPU times (Op×2−Op)	Hardware CPU times (Op×1001−Op)	Speedup
add	vm	101.54±1.6% – 77.91±1.2%	163.50±0.5% – 4.12±0.9%	148×
mul	vm	116.68±1.5% – 77.91±1.2%	163.54±0.5% – 4.12±0.9%	243×
add	native	25.08±2.0% – 20.10±4.8%	88.67±2.2% – 1.66±0.9%	57×
mul	native	29.13±1.2% – 20.10±4.8%	92.79±1.7% – 1.66±0.9%	99×

Benchmarks (3/6) – CoqInterval

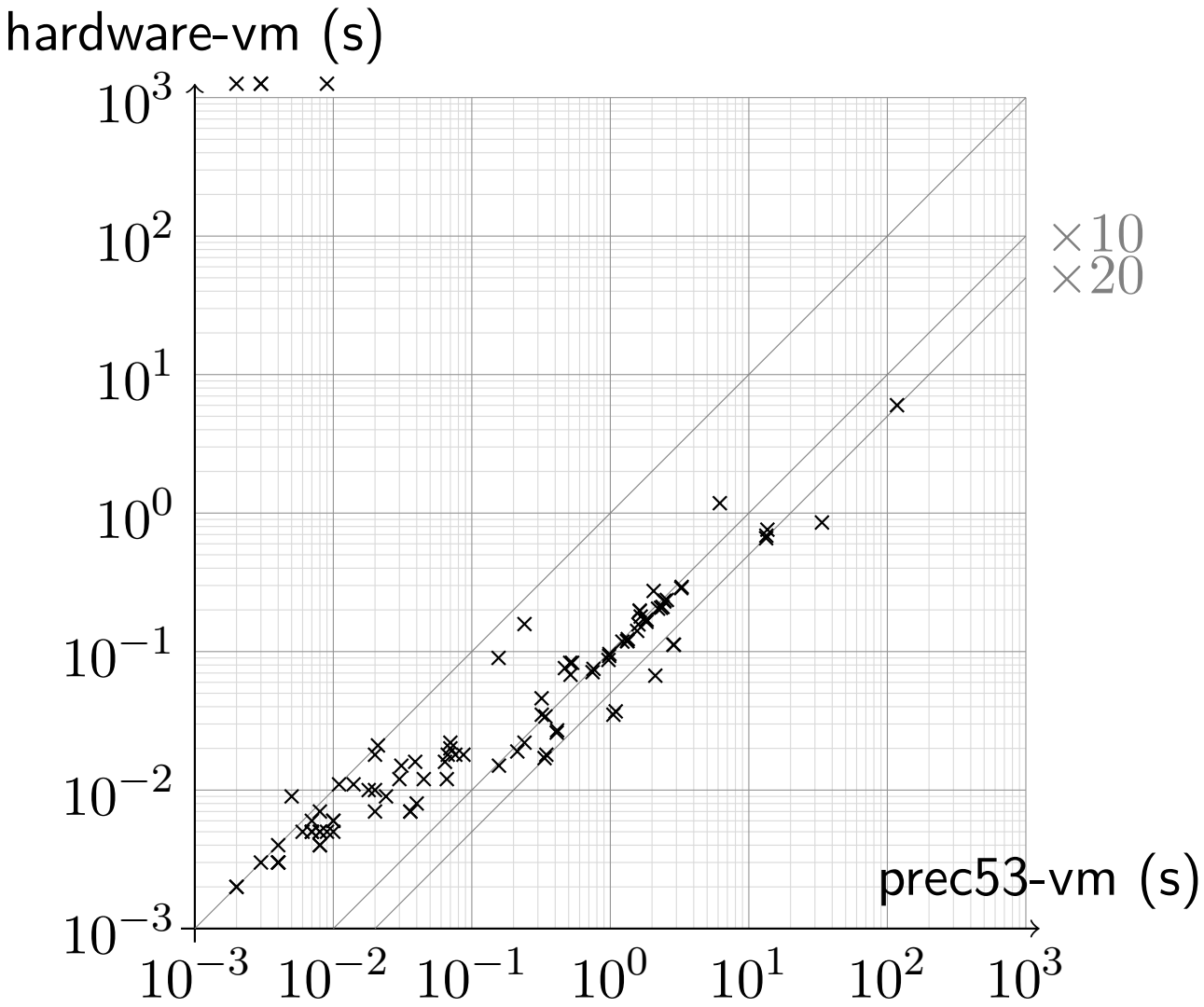


Fig. 1 Comparison of proof times: hardware vs. emulated 53-bit FPA using `vm_compute`

log-log scale (diagonals = equivalent speedups)

Out of the 101 examples, 4 proofs fail with hardware numbers due to the pessimistic outward rounding (points at the top of the graph)

Benchmarks (4/6) – CoqInterval

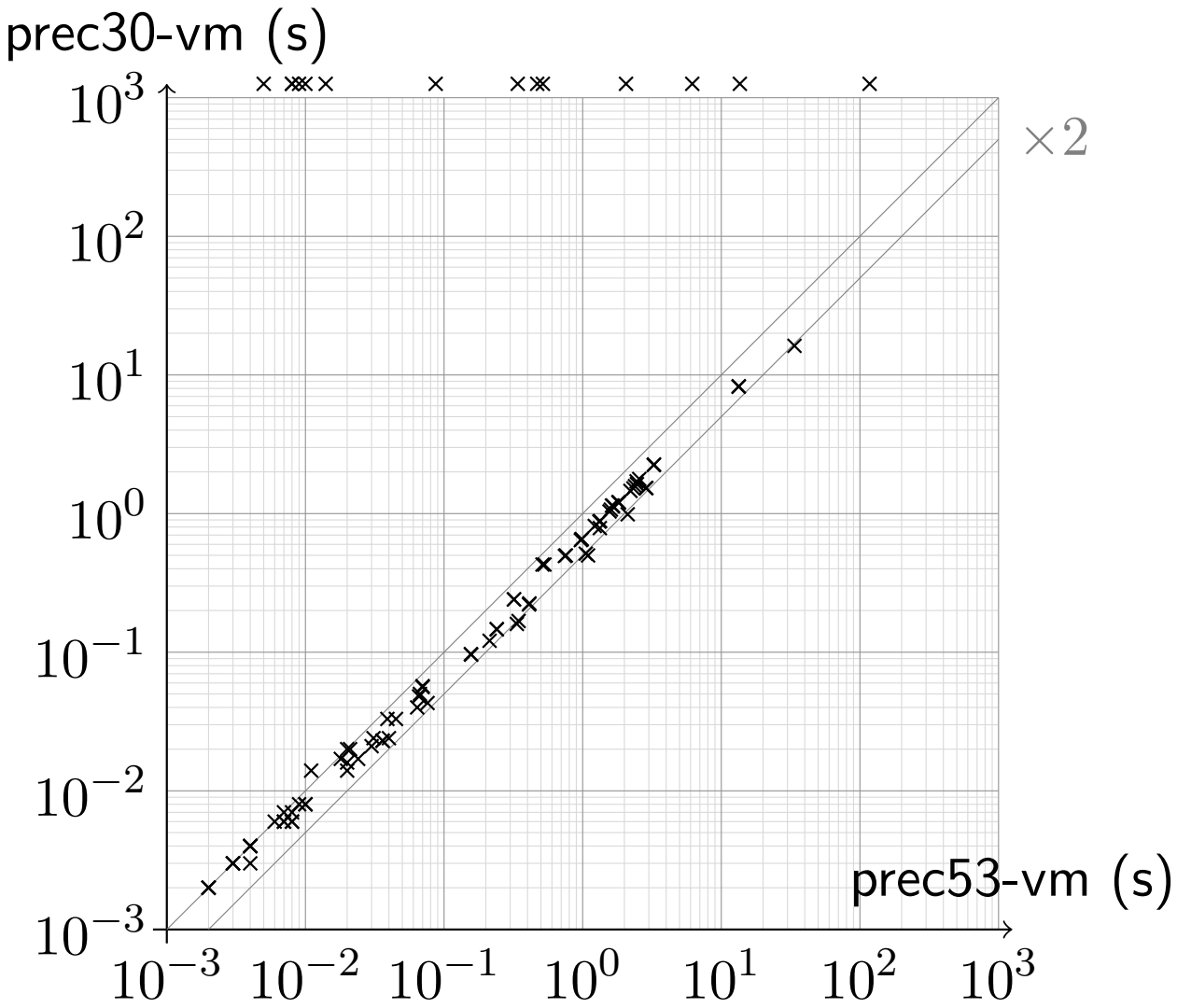


Fig. 2 Comparison of proof times: emulated 53-bit vs. 30-bit FPA using `vm_compute`

Out of the 101 examples, 14 proofs fail with the reduced precision

Before this work, 30 bits = CoqInterval default precision (enough to prove 86% of the suite, but with a smaller speedup wrt. for Fig. 1!)

Benchmarks (5/6) – CoqInterval

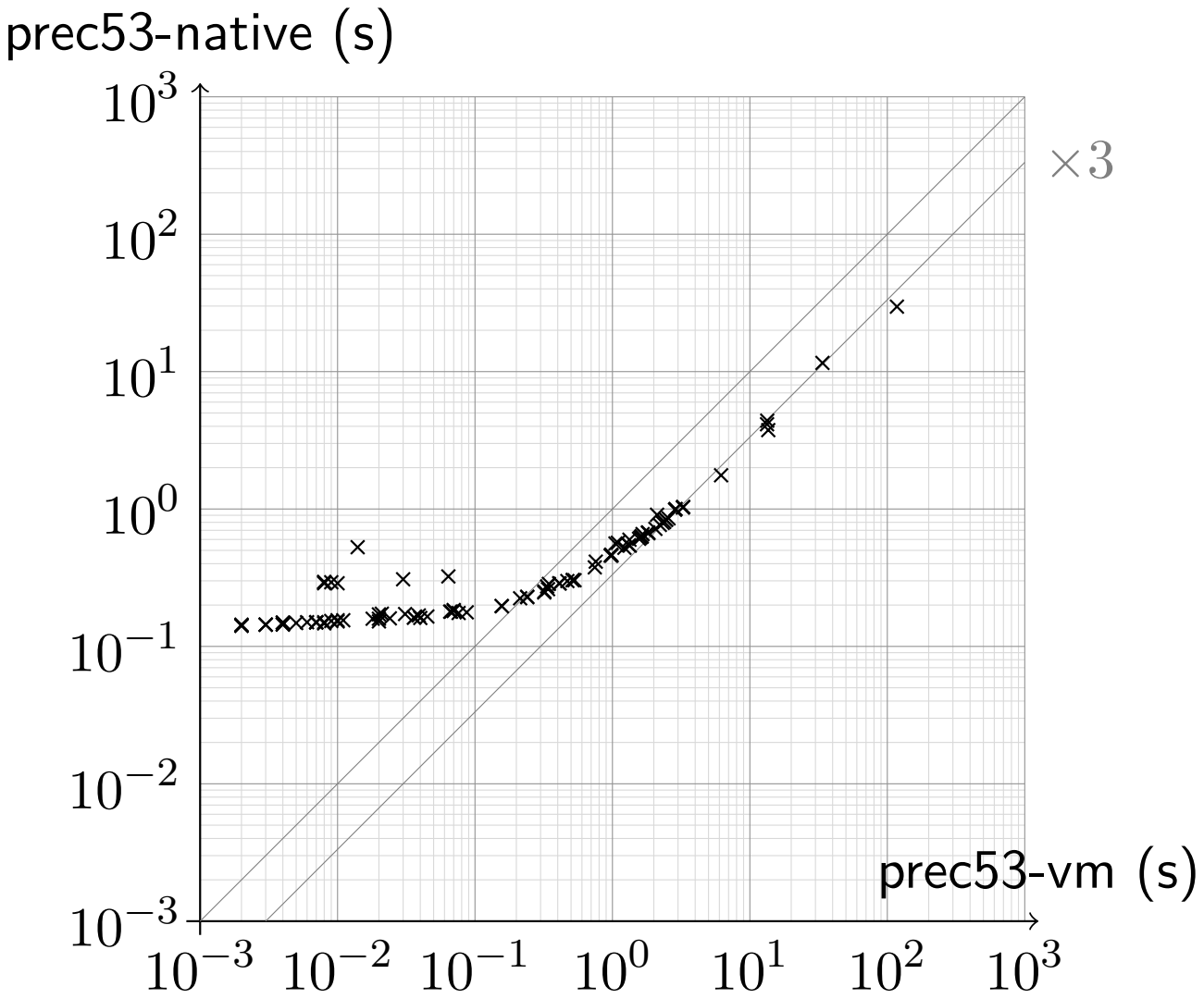


Fig. 3 Comparison of proof times: `vm_compute` vs. `native_compute`, for emulated 53-bit FPA.

Benchmarks (6/6) – CoqInterval

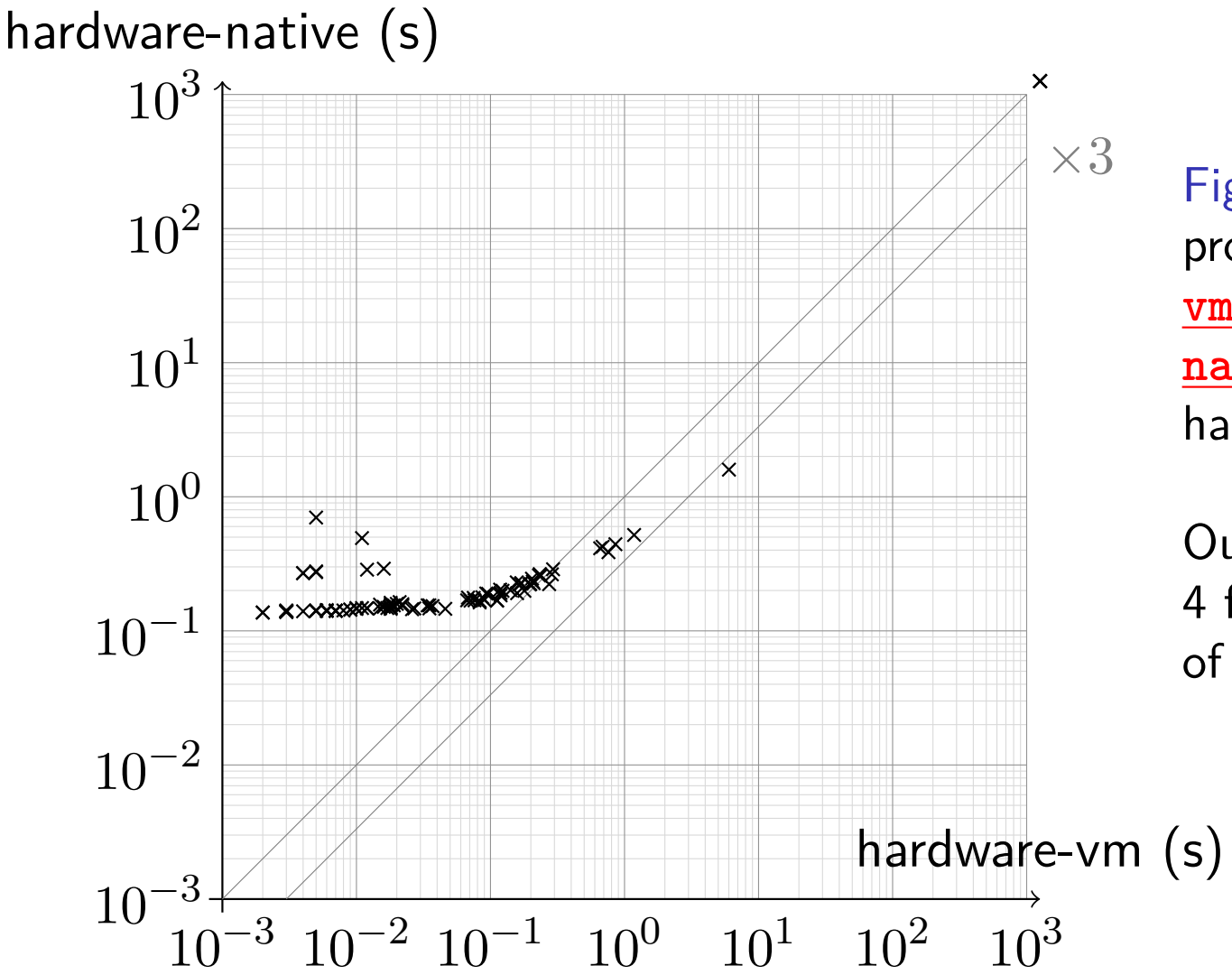


Fig. 4 Comparison of proof times: `vm_compute` vs. `native_compute`, for hardware FPA.

Out of the 101 examples, 4 fail (points at the top of the graph)

Agenda

Concluding remarks

Wrap-up

- Add hardware floating-point support in Coq's stdlib and kernel
- Builds on the methodology of primitive 63-bit integers
- Focus on binary64 and on portability (IEEE 754, no NaN payloads. . .)
- Speedup: 2 OOM for $+/\times$, 1 OOM for reflexive tactics ValidSDP/CoqInterval

Concluding remarks

Wrap-up

- Add hardware floating-point support in Coq's stdlib and kernel
- Builds on the methodology of primitive 63-bit integers
- Focus on binary64 and on portability (IEEE 754, no NaN payloads. . .)
- Speedup: 2 OOM for $+/\times$, 1 OOM for reflexive tactics ValidSDP/CoqInterval

Perspectives

- Implement `roundToIntegral` and `convertToIntegral`?
- Replace unsigned integers with signed integers? (cf. slide 11/26)
- Nice applications (e.g., floating-point expansions?; other ideas?)

Thank you!

```
From Coq Require Import Floats.

(* Questions *)
```

https://coq.github.io/doc/master/refman/language/core/primitive.html#primitive-floats

Proofs involving floating-point computations (1/2)

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 (1/2)

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 (1/2)

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 (2/2)

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 containment 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 (2/2)

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 containment 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.