Computer certified efficient exact reals in Coq

Bas Spitters
Robbert Krebbers
Eelis van der Weegen

Radboud University Nijmegen

Supported by EU FP7 STREP FET-open ForMATH

Why do we need certified exact arithmetic?

- There is a big gap between:
  - Numerical algorithms in research papers.
  - Actual implementations (Mathematica, MATLAB, ...).

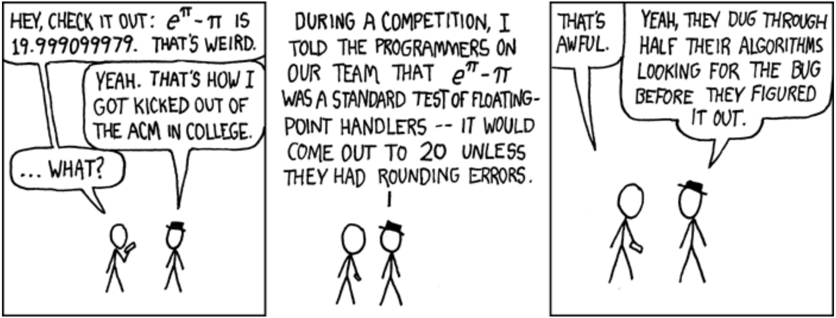
Why do we need certified exact arithmetic?

- There is a big gap between:
  - Numerical algorithms in research papers.
  - Actual implementations (Mathematica, MATLAB, ...).
- This gap makes the code difficult to maintain.
- Makes it difficult to trust the code of these implementations!

Why do we need certified exact arithmetic?

- There is a big gap between:
  - Numerical algorithms in research papers.
  - Actual implementations (MATHEMATICA, MATLAB, ...).
- This gap makes the code difficult to maintain.
- Makes it difficult to trust the code of these implementations!
- Undesirable in proofs that rely on the execution of this code.
  - Kepler conjecture.
  - Existence of the Lorentz attractor.

Why do we need certified exact real arithmetic?



(http://xkcd.com/217/)

Bishop's proposal

Use constructive analysis to bridge this gap.

- Exact real numbers instead of floating point numbers.
- Functional programming instead of imperative programming.
- Dependent type theory.
- A proof assistant to verify the correctness proofs.
- Constructive mathematics to tightly connect mathematics with computations.

Real numbers

- Cannot be represented exactly in a computer.
- Approximation by rational numbers.
- Or any set that is dense in the rationals (e.g. the dyadics).

O'Connor's implementation in Coq

- Based on *metric spaces* and the *completion monad*.

$$\mathbb{R} := \mathfrak{C}\mathbb{Q} := \{f : \mathbb{Q}_+ \to \mathbb{Q} \mid f \textit{ is regular}\}$$

- To define a function $\mathbb{R} \to \mathbb{R}$: define a *uniformly continuous function* $f : \mathbb{Q} \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$.
- Efficient combination of proving and programming.

O'Connor's implementation in Coq

- Based on *metric spaces* and the *completion monad*.

$$\mathbb{R} := \mathfrak{C}\mathbb{Q} := \{f : \mathbb{Q}_+ \to \mathbb{Q} \mid f \textit{ is regular}\}$$

- To define a function $\mathbb{R} \to \mathbb{R}$: define a *uniformly continuous function* $f : \mathbb{Q} \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$.
- Efficient combination of proving and programming.

O'Connor's implementation in Coq

Problem:

- A concrete representation of the rationals (Coq's `Q`) is used.
- Cannot swap implementations, e.g. use machine integers.

O'Connor's implementation in Coq

Problem:

- A concrete representation of the rationals (Coq's `Q`) is used.
- Cannot swap implementations, e.g. use machine integers.

Solution:

Build theory and programs on top of abstract interfaces instead of concrete implementations.

- Cleaner.
- Mathematically sound.
- Can swap implementations.

Our contribution

- Provide an abstract specification of the dense set.
- For which we provide an implementation using the dyadics:

$$n * 2^e \qquad \text{for} \qquad n, e \in \mathbb{Z}$$

- Use Coq's machine integers.
- Extend our algebraic hierarchy based on type classes
- Implement range reductions.
- Improve computation of power series:
  - Keep auxiliary results small.
  - Avoid evaluation of termination proofs.

Interfaces for mathematical structures

- Algebraic hierarchy (groups, rings, fields, ...)
- Relations, orders, ...
- Categories, functors, universal algebra, ...
- Numbers: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, ...

Need solid representations of these, providing:

- Structure inference.
- Multiple inheritance/sharing.
- Convenient algebraic manipulation (e.g. rewriting).
- Idiomatic use of names and notations.

S/and van der Weegen: use type classes

Type classes

- Useful for organizing interfaces of abstract structures.
- Similar to AXIOM's so-called categories.
- Great success in HASKELL and ISABELLE.
- Recently added to COQ.
- Based on already existing features (records, proof search, implicit arguments).

Proof engineering

Similar to canonical structures

Unbundled using type classes

Define *operational type classes* for operations and relations.

```
Class Equiv A := equiv: relation A.
Infix "=" := equiv: type_scope.
Class RingPlus A := ring_plus: A → A → A.
Infix "+" := ring_plus.
```

Represent algebraic structures as predicate type classes.

```
Class SemiRing A {e plus mult zero one} : Prop := {
 semiring_mult_monoid :> @CommutativeMonoid A e mult one ;
 semiring_plus_monoid :> @CommutativeMonoid A e plus zero ;
 semiring_distr :> Distribute (.∗.) (+) ;
 semiring_left_absorb :> LeftAbsorb (.∗.) 0 }.
```

Examples

```
(* z & x = z & y → x = y *)
Instance group_cancel `{Group G} : ∀ z, LeftCancellation (&) z.
```

Examples

```
(* z & x = z & y → x = y *)
Instance group_cancel `{Group G} : ∀ z, LeftCancellation (&) z.

Lemma preserves_inv `{Group A} `{Group B}
 `{!Monoid_Morphism (f : A → B)} x : f (−x) = −f x.
Proof.
 apply (left_cancellation (&) (f x)).
 rewrite ← preserves_sg_op.
 rewrite 2!right_inverse.
 apply preserves_mon_unit.
Qed.
```

Examples

```
(* z & x = z & y → x = y *)
Instance group_cancel `{Group G} : ∀ z, LeftCancellation (&) z.

Lemma preserves_inv `{Group A} `{Group B}
 `{!Monoid_Morphism (f : A → B)} x : f (−x) = −f x.
Proof.
 apply (left_cancellation (&) (f x)).
 rewrite ← preserves_sg_op.
 rewrite 2!right_inverse.
 apply preserves_mon_unit.
Qed.

Lemma cancel_ring_test `{Ring R} x y z : x + y = z + x → y = z.
Proof.
 intros.
 apply (left_cancellation (+) x).
 now rewrite (commutativity x z).
Qed.
```

Number structures

S/van der Weegen specified:

- Naturals: initial semiring.
- Integers: initial ring.
- Rationals: field of fractions of $\mathbb{Z}$.

Basic operations

- Common definitions:
  - nat_pow: repeated multiplication,
  - shiftl: repeated duplication.
- Implementing these operations this way is too slow.
- We want different implementations for different number representations.
- And avoid definitions and proofs becoming implementation dependent.

Basic operations

- Common definitions:
  - nat_pow: repeated multiplication,
  - shiftl: repeated duplication.
- Implementing these operations this way is too slow.
- We want different implementations for different number representations.
- And avoid definitions and proofs becoming implementation dependent.

Hence we want an abstract specification.

Basic operations

- For example:

```
Class ShiftL A B := shiftl: A → B → A.
Infix "≪ " := shiftl (at level 33, left associativity).

Class ShiftLSpec A B (sl : ShiftL A B) `{Equiv A} `{Equiv B}
  `{RingOne A} `{RingPlus A} `{RingMult A}
  `{RingZero B} `{RingOne B} `{RingPlus B} := {
 shiftl_proper : Proper ((=) ==> (=) ==> (=)) (≪) ;
 shiftl_0 :> RightIdentity (≪) 0 ;
 shiftl_S : ∀ x n, x ≪ (1 + n) = 2 ∗ x ≪ n }.
```

Approximate rationals

```
Class AppDiv AQ := app_div : AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx : AQ → Z → AQ.

Class AppRationals AQ {e plus mult zero one inv} ‘{!Order AQ}
  {AQtoQ : Coerce AQ Q_as_MetricSpace} ‘{!AppInverse AQtoQ}
  {ZtoAQ : Coerce Z AQ} ‘{!AppDiv AQ} ‘{!AppApprox AQ}
  ‘{!Abs AQ} ‘{!Pow AQ N} ‘{!ShiftL AQ Z}
  ‘{∀ x y : AQ, Decision (x = y)} ‘{∀ x y : AQ, Decision (x ≤ y)} : Prop := {
  aq_ring :> @Ring AQ e plus mult zero one inv ;
  aq_order_embed :> OrderEmbedding AQtoQ ;
  aq_ring_morphism :> SemiRing_Morphism AQtoQ ;
  aq_dense_embedding :> DenseEmbedding AQtoQ ;
  aq_div : ∀ x y k, B_{2^k}('app_div x y k) ('x / 'y) ;
  aq_approx : ∀ x k, B_{2^k}('app_approx x k) ('x) ;
  aq_shift :> ShiftLSpec AQ Z (≪) ;
  aq_nat_pow :> NatPowSpec AQ N (^) ;
  aq_ints_mor :> SemiRing_Morphism ZtoAQ }.
```

Approximate rationals

Compress

```
Class AppDiv AQ := app_div : AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx : AQ → Z → AQ.
Class AppRationals AQ  ... : Prop := {
  ...
  aq_div : ∀ x y k, B_{2^k}('app_div x y k) ('x / 'y) ;
  aq_approx : ∀ x k, B_{2^k}('app_approx x k) ('x) ;
  ... }
```

- app_approx is used to to keep the size of the numbers "small".
- Define compress := bind (λ ϵ, app_approx x (Qdlog2 ϵ)) such that compress x = x.
- Greatly improves the performance [O'Connor].

Power series

- Well suited for computation if:
  - its coefficients are alternating,
  - decreasing,
  - and have limit 0.
- For example, for $-1 \leq x \leq 0$:

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

- To approximate $\exp x$ with error ε we find a k such that:

$$\frac{x^k}{k!} \leq \varepsilon$$

Power series

Problem: we do not have exact division.

- Parametrize InfiniteAlternatingSum with streams n and d representing the numerators and denominators to postpone divisions.
- Need to find both the length and precision of division.

$$\underbrace{\frac{n_1}{d_1}}_{\frac{\varepsilon}{2k}\text{error}} + \underbrace{\frac{n_2}{d_2}}_{\frac{\varepsilon}{2k}\text{error}} + \ldots + \underbrace{\frac{n_k}{d_k}}_{\frac{\varepsilon}{2k}\text{error}} \qquad \text{such that} \qquad \frac{n_k}{d_k} \leq \varepsilon/2$$

Power series

Problem: we do not have exact division.

- Parametrize InfiniteAlternatingSum with streams n and d representing the numerators and denominators to postpone divisions.
- Need to find both the length and precision of division.

$$\underbrace{\frac{n_1}{d_1}}_{\frac{\varepsilon}{2k}\text{error}} + \underbrace{\frac{n_2}{d_2}}_{\frac{\varepsilon}{2k}\text{error}} + \ldots + \underbrace{\frac{n_k}{d_k}}_{\frac{\varepsilon}{2k}\text{error}} \qquad \text{such that} \qquad \frac{n_k}{d_k} \leq \varepsilon/2$$

- Thus, to approximate $\exp x$ with error ε we need a k such that:

$$\mathbf{B}_{\frac{\varepsilon}{2}} \left(\text{app\_div } n_k \ d_k \ (\log \frac{\varepsilon}{2k}) + \frac{\varepsilon}{2k}\right) 0.$$

Power series

- Computing the length can be optimized using shifts.
- Our approach only requires to compute few extra terms.
- Approximate division keeps the auxiliary numbers “small”.
- We need a trick to avoid evaluation of termination proofs.

What have we implemented so far?

Verified versions of:

- Basic field operations ($+$, $*$, -, $/$)
- Exponentiation by a natural.
- Computation of power series.
- $\exp$, $\arctan$, $\sin$ and $\cos$.
- $\pi := 176 * \arctan\frac{1}{57} + 28 * \arctan\frac{1}{239} - 48 * \arctan\frac{1}{682} + 96 * \arctan\frac{1}{12943}$.
- Square root using Wolfram iteration.

Benchmarks

- Our HASKELL prototype is $\sim$15 times faster.
- Our COQ implementation is $\sim$100 times faster.
- For example:
  - 500 decimals of $\mathsf{exp}\ (\pi * \sqrt{163})$ and $\mathsf{sin}\ (\mathsf{exp}\ 1)$,
  - 2000 decimals of $\mathsf{exp}\ 1000$,

  within 10 seconds in COQ!
- (Previously about 10 decimals)

Benchmarks

- Our HASKELL prototype is $\sim$15 times faster.
- Our COQ implementation is $\sim$100 times faster.
- For example:
  - 500 decimals of $\mathsf{exp}\ (\pi * \sqrt{163})$ and $\mathsf{sin}\ (\mathsf{exp}\ 1)$,
  - 2000 decimals of $\mathsf{exp}\ 1000$,

  within 10 seconds in COQ!
- (Previously about 10 decimals)
- Type classes only yield a 3% performance loss.
- COQ is still too slow compared to unoptimized HASKELL (factor 30 for Wolfram iteration).

Recent improvements

- Verified versions of sin and cos.
- Type class interfaces for constructive {setoids, fields, orders}.
- Additional implementations of AppRationals.
- Avoid evaluation of termination proofs.

Further work

- Newton iteration to compute the square root.
- Geometric series (e.g. to compute log).
- native_compute: evaluation by compilation to OCAML.
  gives COQ $10\times$ boost.
- FLOCQ: more fine grained floating point algorithms.
- Type classified theory on metric spaces.
- What are the benefits of univalence?

Conclusions

- Greatly improved the performance of the reals.
- Abstract interfaces allow to swap implementations and share theory and proofs.
- Type classes yield no apparent performance penalty.
- Nice notations with unicode symbols.

Conclusions

- Greatly improved the performance of the reals.
- Abstract interfaces allow to swap implementations and share theory and proofs.
- Type classes yield no apparent performance penalty.
- Nice notations with unicode symbols.

Issues:

- Type classes are quite fragile.
- Instance resolution is too slow.
- Need to adapt definitions to avoid evaluation in Prop.

Sources

http://robbertkrebbers.nl/research/reals/