Type Classes for Efficient Exact Real Arithmetic in Coq

Robbert Krebbers
Joint work with Bas Spitters\textsuperscript{1}

Radboud University Nijmegen

September 9, 2011 @ TYPES
Bergen, Norway

\textsuperscript{1}The research leading to these results has received funding from the European Union’s 7th Framework Programme under grant agreement nr. 243847 (ForMath).
Why do we need certified exact real arithmetic?

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

- 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.
- Undesirable in safety critical applications.
Why do we need certified exact real 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 real 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.
- Undesirable in safety critical applications.
This talk

Improve performance of real number computation in Coq.
This talk

Improve performance of real number computation in CoQ.

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).
This talk

Improve performance of real number computation in Coq.

**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).

**Coq:**
- Well suited because it is both a dependently typed functional programming language, and,
- a proof assistant for constructive mathematics.
Based on *metric spaces* and the *completion monad*.

\[ \mathbb{R} := \mathcal{CQ} := \{ f : \mathbb{Q}_+ \to \mathbb{Q} \mid f \text{ is regular} \} \]

To define a function \( \mathbb{R} \to \mathbb{R} \): define a *uniformly continuous function* \( f : \mathbb{Q} \to \mathbb{R} \), and obtain \( \tilde{f} : \mathbb{R} \to \mathbb{R} \).

Efficient combination of proving and programming.
O’Connor’s implementation in \( \text{CoQ} \)

**Problem:**

- A concrete representation of the rationals (\( \text{CoQ}'s \ Q \)) 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

An abstract specification of the dense set.

- For which we provide an implementation using the dyadics:
  \[ n \times 2^e \quad \text{for} \quad n, e \in \mathbb{Z} \]

- Using Coq’s machine integers.
- Extend the algebraic hierarchy based on type classes by Spitters and van der Weegen to achieve this.

Some other performance improvements:
- Implement range reductions.
- Improve computation of power series:
  - Keep auxiliary results small.
  - Avoid evaluation of termination proofs.
Our contribution

An abstract specification of the dense set.

- For which we provide an implementation using the dyadics:
  \[ n \cdot 2^e \text{ for } n, e \in \mathbb{Z} \]

- Using Coq's machine integers.
- Extend the algebraic hierarchy based on type classes by Spitters and van der Weegen to achieve this.

Some other performance improvements.

- Implement range reductions.
- Improve computation of power series:
  - Keep auxiliary results small.
  - Avoid evaluation of termination proofs.
Spitters and van der Weegen

Type class based interfaces for:

- A standard algebraic hierarchy.
- Some category theory.
- Some universal algebra.

- Naturals: initial semiring.
- Integers: initial ring.
- Rationals: field of fractions of $\mathbb{Z}$. 
Type class based interfaces for:

- A standard algebraic hierarchy.
- Some category theory.
- Some universal algebra.
- Interfaces for number structures.
  - Naturals: initial semiring.
  - Integers: initial ring.
  - Rationals: field of fractions of $\mathbb{Z}$. 
Our extensions of Spitters and van der Weegen

- Interfaces and theory for operations ($\text{nat}^{\text{pow}}$, $\text{shiftl}$, \ldots).
- Support for undecidable structures.
- Library on constructive order theory (ordered rings, etc.\ldots)
- Explicit casts.
Support for undecidable structures

- To compute $\frac{1}{x}$ for $x \in \mathbb{R}$, one needs a witness $\varepsilon \in \mathbb{Q}_+$ such that $|x| \geq \varepsilon$. 

Cannot be extracted from a proof of $x \neq 0$ because a negation lacks computational content.

Need apartness $\not\sim$ instead of inequality.

1. $\neg x \not\sim x$ (irreflexive)
2. $x \not\sim y \rightarrow y \not\sim x$ (symmetric)
3. $x \not\sim y \rightarrow (x \not\sim z \lor y \not\sim z)$ (co-transitive)
4. $\neg x \not\sim y \leftrightarrow x = y$ (tight)
Support for undecidable structures

To compute $\frac{1}{x}$ for $x \in \mathbb{R}$, one needs a witness $\varepsilon \in \mathbb{Q}^+$ such that $|x| \geq \varepsilon$.

Cannot be extracted from a proof of $x \neq 0$ because a negation lacks computational content.

Need apartness $\preceq$ instead of inequality.

1. $\neg x \preceq x$ (irreflexive)
2. $x \preceq y \rightarrow y \preceq x$ (symmetric)
3. $x \preceq y \rightarrow (x \preceq z \lor y \preceq z)$ (co-transitive)
4. $\neg x \preceq y \leftrightarrow x = y$ (tight)
Apartness in the old version of CoRN

- Informative apartness relation (in \texttt{Type}).
- Easy to extract witnesses.
Apartness in the old version of CoRN

- Informative apartness relation (in $\text{Type}$).
- Easy to extract witnesses.
- Present everywhere in the algebraic hierarchy.
- $\text{CoQ}$ does not support setoid rewriting in $\text{Type}$. 
Apartness in the old version of CoRN

- Informative apartness relation (in \textit{Type}).
- Easy to extract witnesses.
- Present everywhere in the algebraic hierarchy.
- \textit{Coq} does not support setoid rewriting in \textit{Type}.
- \textit{Very heavy in practice}.
Apartness in our development

- Non-informative apartness relation (in Prop).
- Requires additional work to extract witnesses.
Apartness in our development

- Non-informative apartness relation (in \textit{Prop}).
- Requires additional work to extract witnesses.
- Include it just where it is necessary.
- Use type classes to reduce bookkeeping.
Apartness in our development

- Non-informative apartness relation (in Prop).
- Requires additional work to extract witnesses.
- Include it just where it is necessary.
- Use type classes to reduce bookkeeping.
- Easier in practice.
Extracting witnesses

Use constructive indefinite description

**Lemma** `constructive_indefinite_description_nat` (P : nat → Prop):

\[(∀ \ x : nat, \{P x\} + \{¬ P x\}) \rightarrow (∃ \ n : nat, \ P n) \rightarrow \{n : nat \mid P n\}\]

to extract a witness from a Prop-based apartness.
Extracting witnesses

Use constructive indefinite description

Lemma constructive_inddefinite_description_nat (P : nat → Prop) :
(∀ x : nat, {P x} + {¬ P x}) → (∃ n : nat, P n) → {n : nat | P n}

to extract a witness from a Prop-based apartness.

- Performs linear bounded search.
  Slow!
Extracting witnesses

Use constructive indefinite description

Lemma constructive_indefinite_description_nat (P : nat → Prop) :
(∀ x : nat, {P x} + {¬ P x}) → (∃ n : nat, P n) → {n : nat | P n}

to extract a witness from a Prop-based apartness.

- Performs linear bounded search.
  Slow!
- We specify explicit witnesses for computation.
  Faster to obtain, better quality.
We have to look out for cyclic instances, for example

StrongSetoid A \rightarrow Setoid A
Cyclic instances

- We have to look out for cyclic instances, for example

\[
\text{StrongSetoid } A \quad \rightarrow \quad \text{Setoid } A
\]

\[
\text{set } x \leq y := x \neq y, \text{ need decidably equality}
\]
Cyclic instances

► We have to look out for cyclic instances, for example

\[
\text{StrongSetoid } A \quad \rightarrow \quad \text{Setoid } A
\]

\[
\text{set } x \leq y := x \neq y, \text{ need decidably equality makes instance search loop.}
\]

► Create StrongSetoid A from Setoid A instances by hand.
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, B₂ᵏ(’app_div x y k) (’x / ’y) ;
  aq_approx : ∀ x k, B₂ᵏ(’app_approx x k) (’x) ;
  aq_shift :> ShiftLSpec AQ Z (≪) ;
  aq_nat_pow :> NatPowSpec AQ N (^) ;
  aq_ints_mor :> SemiRing_Morphism ZtoAQ }.
Creating the real numbers

- Show that the approximate rationals form a metric space.
- Complete it to obtain the real numbers.
- Lift the ring operations to the real numbers.
- Prove correspondence with O’Connor’s implementation.
Power series

- Well suited for computation if:
  - its coefficients are alternating,
  - decreasing,
  - and have limit 0.

\[
\sin x = \sum_{i=0}^{\infty} \left(\frac{-1}{2i+1}\right)^{2i+1} x^{2i+1}
\]

To approximate \( \sin x \) with error \( \varepsilon \) we find a \( k \) such that:

\[
\left| \left(\frac{-1}{2i+1}\right)^{2i+1} x^{2i+1}\right| \leq \varepsilon
\]
Power series

- Well suited for computation if:
  - its coefficients are alternating,
  - decreasing,
  - and have limit 0.

- For example, for $-1 \leq x \leq 1$:

$$\sin x = \sum_{i=0}^{\infty} (-1)^{i} \cdot \frac{x^{2i+1}}{2i + 1}$$

- To approximate $\sin x$ with error $\varepsilon$ we find a $k$ such that:

$$\left| (-1)^{i} \cdot \frac{x^{2i+1}}{2i + 1} \right| \leq \varepsilon$$
Problem 1: we do not have exact division.

- So, we cannot compute the coefficients \( \frac{x^{2i+1}}{2i+1} \) exactly.
Power series

Problem 1: we do not have exact division.

▶ So, we cannot compute the coefficients \( \frac{x^{2i+1}}{2i+1} \) exactly.
▶ Use 2 streams: numerators and denominators.
Power series

Problem 1: we do not have exact division.
  ▶ So, we cannot compute the coefficients \( \frac{x^{2i+1}}{2i+1} \) exactly.
  ▶ Use 2 streams: numerators and denominators.
  ▶ Need to compute both the length and precision of division.
  ▶ This can be optimized using shifts.
Problem 1: we do not have exact division.

- So, we cannot compute the coefficients $\frac{x^{2i+1}}{2i+1}$ exactly.
- Use 2 streams: numerators and denominators.
- Need to compute both the length and precision of division.
- This can be optimized using shifts.
- Our approach only requires to compute few extra terms.
- Approximate division keeps the auxiliary numbers “small”.
Power series

Problem 2: convince Coq that it terminates.

- Use an inductive proposition to describe limits.

\[\text{Inductive } \text{Exists} \ A \ (P : \text{Stream} \ A \rightarrow \text{Prop}) \ (x : \text{Stream}) : \text{Prop} :=\]
\[| \text{Here} : P \ x \rightarrow \text{Exists} \ P \ x\]
\[| \text{Further} : \text{Exists} \ P \ (\text{tl} \ x) \rightarrow \text{Exists} \ P \ x.\]
Problem 2: convince Coq that it terminates.

- Use an inductive proposition to describe limits.
  \[
  \text{Inductive} \quad \text{Exists} \ A \ (P : \text{Stream} \ A \rightarrow \text{Prop}) \ (x : \text{Stream}) : \text{Prop} := \\
  \quad \mid \text{Here} : P \ x \rightarrow \text{Exists} \ P \ x \\
  \quad \mid \text{Further} : \text{Exists} \ P \ (\text{tl} \ x) \rightarrow \text{Exists} \ P \ x.
  \]

- But, need to make it lazy, otherwise \text{vm\_compute} will evaluate a proposition [O‘Connor].
  \[
  \text{Inductive} \quad \text{LazyExists} \ A \ (P : \text{Stream} \ A \rightarrow \text{Prop}) \ (x : \text{Stream} \ A) : \text{Prop} := \\
  \quad \mid \text{LazyHere} : P \ x \rightarrow \text{LazyExists} \ P \ x \\
  \quad \mid \text{LazyFurther} : (\text{unit} \rightarrow \text{LazyExists} \ P \ (\text{tl} \ x)) \rightarrow \text{LazyExists} \ P \ x.
  \]
Unfortunately, still too much overhead.

- Perform 50,000 steps before looking at the proof.

```ocaml
Fixpoint LazyExists_inc ‘{P : Stream A → Prop}
  (n : nat) s : LazyExists P (Str_nth_tl n s) → LazyExists P s :=
  match n return LazyExists P (Str_nth_tl n s) → LazyExists P s with
  | O ⇒ λ x, x
  | S n ⇒ λ ex, LazyFurther (λ _, LazyExists_inc n (tl s) ex)
  end.
```
Unfortunately, still too much overhead.

- Perform 50,000 steps before looking at the proof.

```ocaml
Fixpoint LazyExists_inc \{P : Stream A \rightarrow Prop\} 
  (n : nat) s : LazyExists P (Str_nth_tl n s) \rightarrow LazyExists P s := 
match n return LazyExists P (Str_nth_tl n s) \rightarrow LazyExists P s with 
    | O  ⇒ λ x, x 
    | S n ⇒ λ ex, LazyFurther (λ _, LazyExists_inc n (tl s) ex) 
end.
```

- Major ($\geq$ 10 times) performance improvement!
We extend the sine to its complete domain by repeatedly applying:

\[
\sin x = 3 \sin \left( \frac{x}{3} \right) - 4 \left( \sin \left( \frac{x}{3} \right) \right)^3
\]
We extend the sine to its complete domain by repeatedly applying:

$$\sin x = 3 \times \sin \frac{x}{3} - 4 \times \left( \sin \frac{x}{3} \right)^3$$

Efficient because we postpone divisions.
Extending the sine to its complete domain

- We extend the sine to its complete domain by repeatedly applying:

\[ \sin x = 3 \cdot \sin \left( \frac{x}{3} \right) - 4 \cdot \left( \sin \left( \frac{x}{3} \right) \right)^3 \]

- Efficient because we postpone divisions.
- Performance improves significantly by reducing the input to a value between \(-2^k \leq x \leq 0\) for \(50 \leq k\).
- Faster than subtracting multiples of \(2\pi\) because our implementation of \(\pi\) is too slow.
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.
- \( \pi := 176 \cdot \arctan \frac{1}{57} + 28 \cdot \arctan \frac{1}{239} - 48 \cdot \arctan \frac{1}{682} + 96 \cdot \arctan \frac{1}{12943} \).
- Square root using Wolfram iteration.
Benchmarks

- Our **Haskell** prototype is \(~15\) times faster.
- Our **Coq** implementation is \(~100\) times faster.
- For example:
  - 500 decimals of \(\exp \left( \pi \times \sqrt{163} \right)\) and \(\sin \left( \exp 1 \right)\),
  - 2000 decimals of \(\exp 1000\),
  - 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 \( \exp(\pi \times \sqrt{163}) \) and \( \sin(\exp 1) \),
  - 2000 decimals of \( \exp 1000 \),
  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).
Further work

- Newton iteration to compute the square root.
- Geometric series (e.g. to compute $\ln$).
- `native_compute`: evaluation by compilation to OCAML.
- `FLOCQ`: more fine grained floating point algorithms.
- Type classified theory on metric spaces.
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 names and notations with type classes and unicode symbols.
Issues

- Type classes are quite fragile.
- Instance resolution is too slow.
- Instance resolution cannot handle cyclic instances.
- No setoid rewriting in for relations in `Type`.
- Need to adapt definitions to avoid evaluation in `Prop`.
Sources

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