Type Classes for Efficient Exact Real Arithmetic in Coq

Robbert Krebbers
Joint work with Bas Spitters[1]

Radboud University Nijmegen

September 9, 2011 @ TYPES
Bergen, Norway

[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, ...).

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.

Starting point: O'Connor's implementation in Coq

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

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

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

An abstract specification of the dense set.

- For which we provide an implementation using the dyadics:

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

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

Our contribution

An abstract specification of the dense set.

- For which we provide an implementation using the dyadics:

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

- 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.

Spitters and van der Weegen

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 (nat_pow, shiftl, ...).
- Support for undecidable structures.
- Library on constructive order theory (ordered rings, etc...)
- 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 $|x| \geq \varepsilon$.

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 $\lessgtr$ instead of inequality.
  1. $\neg x \lessgtr x$ (irreflexive)
  2. $x \lessgtr y \to y \lessgtr x$ (symmetric)
  3. $x \lessgtr y \to (x \lessgtr z \vee y \lessgtr z)$ (co-transitive)
  4. $\neg x \lessgtr y \leftrightarrow x = y$ (tight)

Apartness in the old version of CoRN

- Informative apartness relation (in `Type`).
- Easy to extract witnesses.

Apartness in the old version of CoRN

- Informative apartness relation (in Type).
- Easy to extract witnesses.
- Present everywhere in the algebraic hierarchy.
- Coq does not support setoid rewriting in Type.

Apartness in the old version of CoRN

- Informative apartness relation (in Type).
- Easy to extract witnesses.
- Present everywhere in the algebraic hierarchy.
- Coq does not support setoid rewriting in Type.
- 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 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}) → (∃ n : nat, P n) → {n : nat | P n}
```

to extract a witness from a Prop-based apartness.

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!

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.

Cyclic instances

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

$$\begin{tikzcd} \text{StrongSetoid A} \arrow[rr, bend left, Rightarrow] & & \text{Setoid A} \end{tikzcd}$$

Cyclic instances

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

$$\begin{tikzcd}
\text{StrongSetoid A} \arrow[rr, Rightarrow, bend left] & & \text{Setoid A} \arrow[ll, Rightarrow, bend left, "\text{set } x \lessgtr y := x \neq y\text{, need decidably equality}"]
\end{tikzcd}$$

Cyclic instances

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

$$
\begin{tikzcd}
\text{StrongSetoid A} \arrow[rr, Rightarrow, bend left] & & \text{Setoid A} \arrow[ll, Rightarrow, bend left, "\text{set } x \lessgtr y := x \neq y\text{, need decidably equality}"]
\end{tikzcd}
$$

  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, 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 }.
```

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.

Power series

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

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

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

$$\left| (-1)^i * \frac{x^{2i+1}}{2i+1} \right| \leq \varepsilon$$

Power series

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.

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.
- 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.

```
Inductive Exists A (P : Stream A → Prop) (x : Stream) : Prop :=
  | Here : P x → Exists P x
  | Further : Exists P (tl x) → Exists P x.
```

Power series

Problem 2: convince Coq that it terminates.

- Use an inductive proposition to describe limits.

```
Inductive Exists A (P : Stream A → Prop) (x : Stream) : Prop :=
  | Here : P x → Exists P x
  | Further : Exists P (tl x) → Exists P x.
```

- But, need to make it lazy, otherwise vm_compute will evaluate a proposition [O'Connor].

```
Inductive LazyExists A (P : Stream A → Prop) (x : Stream A) : Prop :=
  | LazyHere : P x → LazyExists P x
  | LazyFurther : (unit → LazyExists P (tl x)) → LazyExists P x.
```

Power series

Unfortunately, still too much overhead.

- Perform 50.000 steps before looking at the proof.

```
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.
```

Power series

Unfortunately, still too much overhead.

- Perform 50.000 steps before looking at the proof.

```
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.
```

- Major ($\geq$ 10 times) performance improvement!

Extending the sine to its complete domain

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

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

Extending the sine to its complete domain

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

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

- Efficient because we postpone divisions.

Extending the sine to its complete domain

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

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

- 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 2π because our implementation of π 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.
- $\pi := 176 * \mathsf{arctan}\frac{1}{57} + 28 * \mathsf{arctan}\frac{1}{239} - 48 * \mathsf{arctan}\frac{1}{682} + 96 * \mathsf{arctan}\frac{1}{12943}$.
- 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 $\mathsf{exp}\ (\pi * \sqrt{163})$ 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 $\exp(\pi * \sqrt{163})$ 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/