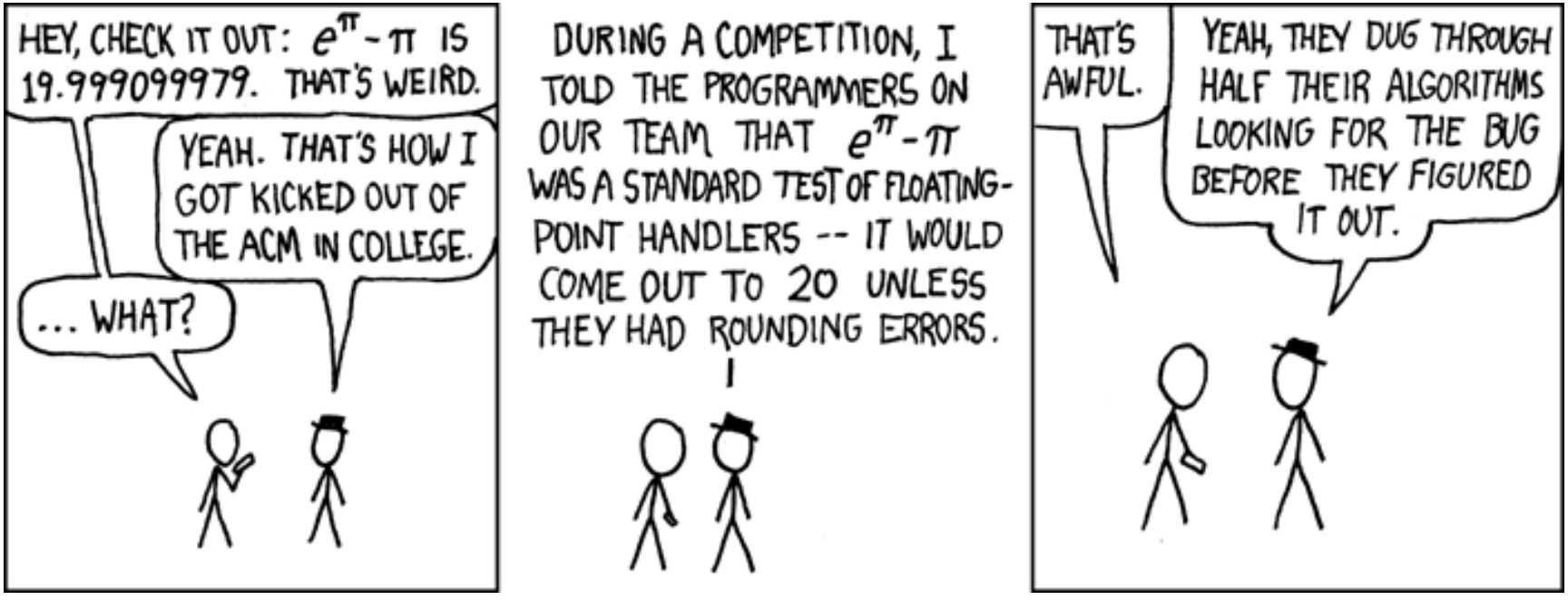
Computer certified efficient exact reals in Coq

Robbert Krebbers
Joint work with Bas Spitters

Radboud University Nijmegen

March 22, 2011

Why do we need certified exact reals?



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

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*.

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

- 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.

But unfortunately:

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

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 the algebraic hierarchy based on type classes by Spitters and van der Weegen.
- Improve computation of power series using approximate division.

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.

Spitters and van der Weegen: use type classes!

Fully unbundled

```
Definition reflexive {A: Type} (R : relation A) : Prop := ∀ a, R a a.
```

Flexible in theory, inconvenient in practice:

- Nothing to bind notations to
- Declaring/passing inconvenient
- No structure inference

Fully bundled

```
Record SemiGroup : Type := {
 sg_car :> Setoid ;
 sg_op : sg_car → sg_car → sg_car ;
 sg_proper : Proper ((=) ⟹ (=) ⟹ (=)) sg_op ;
 sg_ass : ∀ x y z, sg_op x (sg_op y z) = sg_op (sg_op x y) z) }
```

Problems:

- Prevents sharing, e.g. group together two CommutativeMonoids to create a SemiRing.
- Multiple inheritance (diamond problem).
- Long projection paths.

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.

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

Spitters and 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 multiplication by 2.
- 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.

Abstract specifications

Using Σ-types

- Well suited for simple functions.
- An example:

  Class Abs A ‘{Equiv A} ‘{Order A} ‘{RingZero A} ‘{GroupInv A}
  $:=$ abs_sig: $\forall$ x, { y | $(0 \leq x \rightarrow y = x) \wedge (x \leq 0 \rightarrow y = -x)$}.
  Definition abs ‘{Abs A} $:= \lambda$ x : A, ‘ (abs_sig x).

- Program allows to create instances easily.

  Program Instance: Abs Z := Zabs.

- But unable to quantify over all possible input values.

Abstract specifications

Bundled

- For example:

```
Class ShiftL A B `{Equiv A} `{Equiv B} `{RingOne A} `{RingPlus A}
  `{RingMult A} `{RingZero B} `{RingOne B} `{RingPlus B} := {
 shiftl : A → B → A ;
 shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) shiftl ;
 shiftl_0 :> RightIdentity shiftl 0 ;
 shiftl_S : ∀ x n, shiftl x (1 + n) = 2 * shiftl x n }.
Infix "≪" := shiftl (at level 33, left associativity).
```

- Here shiftl is a δ-redex, hence simpl unfolds it.
- For BigN, x ≪ n becomes BigN.shiftl x n.
- As a result, rewrite often fails.

Basic operations

Unbundled

- 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 }.
```

- The δ-redex is gone due to the operational class.
- Remark: not `shiftl x n := x * 2 ^ n` since we cannot take a negative power on the dyadics.

Theory on basic operations

- Theory on shifting with exponents in $\mathbb{N}$ and $\mathbb{Z}$ is similar.
- Want to avoid duplication of theorems and proofs.

```
Class Biinduction R `{Equiv R}
  `{RingZero R} `{RingOne R} `{RingPlus R} : Prop
 := biinduction (P: R → Prop) `{!Proper ((=) ⟹ iff) P} :
       P 0 → (∀ n, P n ↔ P (1 + n)) → ∀ n, P n.
```

- Some syntax:

```
Section shiftl.
 Context `{SemiRing A} `{!LeftCancellation (.*.) (2:A)}
   `{SemiRing B} `{!Biinduction B} `{!ShiftLSpec A B sl}.

 Lemma shiftl_base_plus x y n : (x + y) ≪ n = x ≪ n + y ≪ n.

 Global Instance shiftl_inj: ∀ n, Injective (≪n).
End shiftl.
```

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, $\mathbf{B}_{2^k}$('app_div x y k) ('x / 'y) ;
aq_approx : ∀ x k, $\mathbf{B}_{2^k}$('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 1: convince Coq that this 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

Problem 2: 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.

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

- 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”.

Extending the exponential to its complete domain

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

$$\exp\, x = (\exp\, (x \ll 1))^2$$

- Performance improves significantly by reducing the input to a value between $-2^k \leq x \leq 0$ for $50 \leq k$.

What have we implemented so far?

Verified versions of:

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

Benchmarks

- Our Haskell prototype is ∼15 times faster.
- Our Coq implementation is ∼100 times faster.
- Now able to compute 2,000 decimals of π and 425 decimals of $\exp \pi - \pi$ within one minute in Coq!
- (Previously 300 and 25 decimals)
- Type classes only yield a 3% performance loss.
- Coq is still too slow compared to unoptimized Haskell (factor 30 for Wolfram iteration).

Improvements

- Newton iteration to compute the square root.
- 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 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/