The Picard Algorithm for Ordinary Differential Equations in Coq

Bas Spitters

VALS - LRI

23 May 2014
ForMath 2010-2013

Why do we need certified exact arithmetic?

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

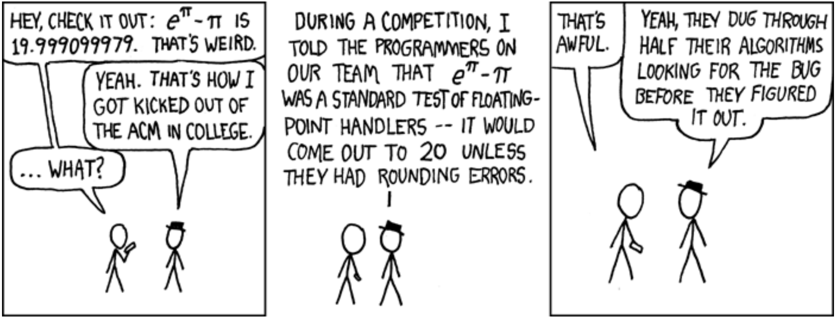
Why do we need certified exact arithmetic?

- 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/)

I also hear that $9^2 + 19^2/22 = \pi^4$.

Strategy

- Exact real numbers instead of floating point numbers.
- Functional programming instead of imperative programming.
- Dependent type theory.
- A proof assistant (Coq) to verify the correctness proofs.
- Constructive analysis 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).
- No computable zero-test:
  0,000

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).
- No computable zero-test:
  0,00000000000000000000000

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).
- No computable zero-test:
  0,00000000000000000000000000000000000000

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).
- No computable zero-test:
  0,000000000000000000000000000000000000001

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).
- No computable zero-test:
  0,00000000000000000000000000000000000000001
  Under Curry-Howard correspondence not:

$$x = 0 \vee \neg x = 0$$

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).
- No computable zero-test:
  0,00000000000000000000000000000000000000001
  Under Curry-Howard correspondence not:

  $$x = 0 \vee \neg x = 0$$

  Do have: for every $\epsilon > 0$

  $$|x| \leq \epsilon \vee |x| > 0$$

O'Connor's implementation in Coq

Refined Cauchy sequences:

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

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

  Idea: $f(\epsilon)$ is an ϵ-approximation to the real number.
- To define a function $\mathbb{R} \to \mathbb{R}$: define a *uniformly continuous function* $f : \mathbb{Q} \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$.
  Monadic programming like in haskell.
- Efficient combination of proving and programming.

O'Connor's implementation in Coq

Problem:

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

O'Connor's implementation in Coq

Problem:

- A concrete representation of the rationals (`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.

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.
- Added to COQ by Oury and Souzeau,
  Based on already existing features (records, proof search, implicit arguments).

Proof engineering

Comparison(?) to canonical structures, unification hints

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 }.
```

Separate structure and property from category theory.

Naturals as Semiring

```
Instance nat_equiv: Equiv nat := eq.
Instance nat_plus: Plus nat := Peano.plus.
Instance nat_0: Zero nat := 0%nat.
Instance nat_1: One nat := 1%nat.
Instance nat_mult: Mult nat := Peano.mult.

Instance: SemiRing nat.
Proof.
  repeat (split; try apply _); repeat intro.
              now apply plus_assoc.
             now apply plus_0_r.
            now apply plus_comm.
           now apply mult_assoc.
          now apply mult_1_l.
         now apply mult_1_r.
        now apply mult_comm.
  now apply mult_plus_distr_l.
Qed.
```

Number structures

S/van der Weegen specified:

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

Building on a library for basic cat th and universal algebra.

Number structures

S/van der Weegen specified:

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

Building on a library for basic cat th and universal algebra.

Slightly pedantic?

Speeding up

- 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.

Krebbers/S

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 }.
```

Instances of Approximate Rationals

Representation $\texttt{mant} \cdot 2^{\texttt{expo}}$:

```
Record Dyadic Z := dyadic { mant: Z; expo: Z }.

Instance dy_mult: Mult Dyadic :=
  λ x y, dyadic (mant x ∗ mant y) (expo x + expo y).

Instance : AppRationals (Dyadic bigZ).

Instance : AppRationals bigQ.

Instance : AppRationals Q.
```

Similar to floqc.

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.

Efficient Reals

In CoRN, `MetricSpace` is a regular `Record`, not a type class.

```
Coq < Check Complete.
Complete : MetricSpace -> MetricSpace

Coq < Check Q_as_MetricSpace.
Q_as_MetricSpace : MetricSpace

Coq < Check AQ_as_MetricSpace.
AQ_as_MetricSpace :
  ∀ (AQ : Type)  . . . , AppRationals AQ -> MetricSpace

Coq < Definition CR := Complete Q_as_MetricSpace.

Coq < Definition AR := Complete AQ_as_MetricSpace.
```

`AR` is an instance of `Zero`, `Plus`, `Le`, `Field`, `FullPseudoSemiRingOrder`, etc., from the MathClasses library.

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.

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

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.

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

- 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”.
- We use a method to avoid evaluation of termination proofs.

What have we implemented?

Verified versions of:

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

Benchmarks

Compared to O'Connor

- 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

Compared to O'Connor

- Our HASKELL prototype is ~15 times faster.
- Our COQ implementation is ~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).

Future work on real computation

- Newton iteration to compute the square root.
- native_compute: evaluation by compilation to OCAML.
  gives COQ $10\times$ boost.
- FLOCQ: more fine grained floating point algorithms.
- Parametricity combined with type classes?
  Cohen, Dénès, Mortberg

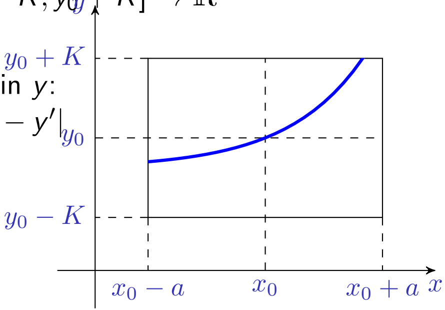
Picard-Lindelöf Theorem

Consider the initial value problem

$$y'(x) = v(x, y(x)), \quad y(x_0) = y_0$$

where

- $v : [x_0 - a, x_0 + a] \times [y_0 - K, y_0 + K] \to \mathbb{R}$
- v is continuous
- v is Lipschitz continuous in y:
  $|v(x, y) - v(x, y')| \leq L|y - y'|$
  for some $L > 0$
- $|v(x, y)| \leq M$
- $aL < 1$
- $aM \leq K$



Such problem has a unique solution on $[x_0 - a, x_0 + a]$.

Proof Idea

$$y'(x) = v(x, y(x)), \quad y(x_0) = y_0$$

is equivalent to

$$y(x) = y(x_0) + \int_{x_0}^{x} v(t, y(t))\,dt$$

Define

$$\begin{aligned}(Tf)(x) &= y_0 + \int_{x_0}^{x} F(t, f(t))\,dt \\ f_0(x) &= y_0 \\ f_{n+1} &= Tf_n\end{aligned}$$

Under the assumptions, T is a contraction on the metric space

$$C([x_0 - a, x_0 + a], [y_0 - K, y_0 + K]).$$

By the Banach fixpoint theorem, T has a fixpoint f and $f_n \to f$.

Metric Spaces

Let (X, d) where $d : X \to X \to \mathbb{R}$ be a metric space.

Let $Brxy$ denote $d(x, y) \leq r$.

A function $f : \mathbb{Q}^+ \to X$ is called regular if
$\forall \varepsilon_1\ \varepsilon_2 : \mathbb{Q}^+,\ B(\varepsilon_1 + \varepsilon_2)(f\varepsilon_1)(f\varepsilon_2)$.

The completion $\mathfrak{C}X$ of X is the set of regular functions.

Let X and Y be metric spaces. A function $f : X \to Y$ is called uniformly continuous with modulus μ if
$\forall \varepsilon : \mathbb{Q}^+\ \forall x_1\ x_2 : X,\ B(\mu\varepsilon)x_1x_2 \to B\varepsilon(fx_1)(fx_2)$.

If $x_1, x_2 : \mathfrak{C}X$, let $B_{\mathfrak{C}X}\varepsilon x_1 x_2 := \forall \varepsilon_1 \varepsilon_2 : \mathbb{Q}^+,\ B_X(\varepsilon_1 + \varepsilon + \varepsilon_2)(x_1\varepsilon_1)(x_2\varepsilon_2)$

Metric spaces with uniformly continuous functions form a category.

Completion forms a monad in the category of metric spaces and uniformly continuous functions.

Completion as a Monad

$$\text{unit} : X \to \mathfrak{C}\,X := \lambda x \lambda \varepsilon,\ x$$

$$\text{join} : \mathfrak{C}\,\mathfrak{C}\,X \to \mathfrak{C}\,X := \lambda x \lambda \varepsilon,\ x(\varepsilon/2)(\varepsilon/2)$$

$$\text{map} : (X \to Y) \to (\mathfrak{C}\,X \to \mathfrak{C}\,Y) := \lambda f \lambda x,\ f \circ x \circ \mu_f$$

$$\text{bind} : (X \to \mathfrak{C}\,Y) \to (\mathfrak{C}\,X \to \mathfrak{C}\,Y) := \text{join} \circ \text{map}$$

Define functions $\mathbb{Q} \to \mathfrak{C}\,\mathbb{Q}$; lift to $\mathfrak{C}\,\mathbb{Q} \to \mathfrak{C}\,\mathbb{Q}$.

Integral

Following Bridger, *Real Analysis: A Constructive Approach*.

```
Class Integral (f: Q -> CR) :=
  integrate: forall (from: Q) (w: QnonNeg), CR.

Notation "∫" := integrate.

Class Integrable `{!Integral f}: Prop := {
  integral_additive:
    forall (a: Q) b c, ∫f a b + ∫f (a + b) c == ∫f a (b + c);

  integral_bounded_prim: forall (from: Q) (width: Qpos) (mid: Q) (r: Qpos),
    (forall x, from <= x <= from + width -> ball r (f x) mid) ->
    ball (width * r) ∫( f from width) (width * mid);

  integral_wd :>
    Proper (Qeq ==> QnonNeg.eq ==> @st_eq CRasCSetoid) ∫( f)
}.
```

Earlier (abstract, but slower) implementation of integral by O'Connor/S.
Implemented in Coq: Makarov/S.

Complexity

Rectangle rule:

$$\left| \int_a^b f(x)\,dx - f(a)(b-a) \right| \leq \frac{(b-a)^3}{24} M$$

where $|f''(x)| \leq M$ for $a \leq x \leq b$.

Number of intervals to have the error $\leq \varepsilon$: $\geq \sqrt{\dfrac{(b-a)^3 M}{24\varepsilon}}$

Simpson's rule:

$$\left| \int_a^b f(x)\,dx - \frac{b-a}{6} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) \right| \leq \frac{(b-a)^5}{2880} M$$

where $|f^{(4)}(x)| \leq M$ for $a \leq x \leq b$.

Coquand/S. *A constructive proof of Simpson's Rule*, replacing differentiation with integration.
The number of points grows exponentially with the number of significant digits.

Future Work

Change the development from CR to AR based on dyadic rationals.

Implement Simpson's integration and prove its error bounds.

Compute forward instead of backward? ($\mathbb{R}$RAM)?

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.

Challenges of current Coq

For discrete mathematics the ssreflect machinery works very well!
The extension to infinitary mathematics is challenging.
No quotients, functional extensionality, subsets, ...
Univalence axiom provides a uniform solution.

Univalence and analysis?

Challenges of current Coq

For discrete mathematics the ssreflect machinery works very well!
The extension to infinitary mathematics is challenging.
No quotients, functional extensionality, subsets, ...
Univalence axiom provides a uniform solution.

Univalence and analysis?

Homotopy type theory

New foundation for mathematics, developed by group of researchers at Princeton.
New type theory: funext, subsets, quotients, unique choice, proof irrelevance (hprop), K-axiom (hsets), ...

Prototype implementation (Coquand et al).
Axiomatic prototypes in Coq/agda.

Use for the reals

Ssreflect Boolean reflection does not directly extend to the reals.

With univalence, we can start to do this: $\mathbb{S}$ reflection.
Sierpinski space $\mathbb{S}$, $1_\perp$, a quotient, classifies the opens/semidecidable propositions.

Use for the reals

Ssreflect Boolean reflection does not directly extend to the reals.

With univalence, we can start to do this: $\mathbb{S}$ reflection.
Sierpinski space $\mathbb{S}$, $1_\perp$, a quotient, classifies the opens/semidecidable propositions.

Structure Invariance Principle (SIP):
Isomorphic structures may be identified: unary and binary numbers, integers, rationals, ...
They may be identified, as types, semirings, ...

Use for the reals

Ssreflect Boolean reflection does not directly extend to the reals.

With univalence, we can start to do this: $\mathbb{S}$ reflection.
Sierpinski space $\mathbb{S}$, $1_\perp$, a quotient, classifies the opens/semidecidable propositions.

Structure Invariance Principle (SIP):
Isomorphic structures may be identified: unary and binary numbers, integers, rationals, ...
They may be identified, as types, semirings, ...

Parametricity: have two representations:
for computing: Cauchy
one for proving: the quotient.