Computer certified efficient exact reals in Coq

Robbert Krebbers and Bas Spitters[⋆]

Radboud University Nijmegen

Abstract. Floating point operations are fast, but require continuous effort on the part of the user in order to ensure that the results are correct. This burden can be shifted away from the user by providing a library of *exact* analysis in which the computer handles the error estimates. We provide an implementation of the exact real numbers in the Coq proof assistant. This improves on the earlier Coq-implementation by O'Connor in two ways: we use dyadic rationals built from the machine integers and we optimize computation of power series by using approximate division. Moreover, we use type classes for clean mathematical interfaces. This appears to be the first time that type classes are used in heavy computation. We obtain over a 100 times speed up of the basic operations and indications for improving the Coq system.

1 Introduction

Real numbers cannot be represented exactly in a computer. Hence, in constructive analysis [1] one approximates real numbers by rational, or dyadic numbers. The real numbers are the completion of the rationals. This completion construction can be organized in a monad, a familiar construct from functional programming (Section 3). The completion monad provides an efficient combination of proving and computing [2]. In this way, O'Connor [3] implements exact real numbers and the transcendental functions on them in Coq.

A number of possible improvements in this implementation were already suggested in [4]. First, we can use Coq's new machine integers; see Section 2. Second, we can use dyadic rationals (that are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$, also known as infinitary floats). Third, the implementation of power series can be improved by using approximate division. Here we carry out all three optimizations. Unfortunately, changing O'Connor's implementation to use the new machine integers was far for from trivial, as he used a particular concrete representation of the rationals. To avoid this in the future, we provide an abstract specification of the dense set as *approximate rationals*; see Section 4.

In Section 4 we provide some abstract order theory culminating in the theory of approximate rationals. Section 5 deals with computing power series using dyadics. Section 6 describes Wolfram's algorithm to compute the square root of a real number. We finish with some benchmarks in Section 7.

[⋆] The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

2 The Coq-system

The Coq proof assistant is based on the calculus of inductive constructions [5, 6], a dependent type theory with (co)inductive types; see [7, 8]. In true Curry-Howard fashion, it is both a pure functional programming language with an expressive type system, and a language for mathematical statements and proofs. We highlight some aspects of Coq relevant for our development.

Types and propositions. Propositions in Coq are types [9, 10], which themselves have types called *sorts*. Coq features a distinguished sort called Prop that one may choose to use as the sort for types representing propositions. The distinguishing feature of the Prop sort is that terms of non-Prop type may not depend on the values of inhabitants of Prop types (that is, proof terms). This regime of discrimination establishes a weak form of proof irrelevance, in that changing a proof can never affect the result of value computations. On a practical level, this lets Coq safely erase all Prop components when extracting certified programs to Ocaml or Haskell. We should note however, that in practice, Coq's extraction mechanism is still very hard to use for programs with the complexity, in terms of depth of definitions, that we are interested in.

Equality, setoids, and rewriting Because the Coq type theory lacks quotient types (as it would make type checking undecidable), one usually bases abstract structures on a *setoid* ('Bishop set'): a type equipped with an equivalence relation [1, 11]. This leads to a naive set theory as described by Palmgren [12]. When the user attempts to substitute a given (sub)term using an equality, the system keeps track of, resolves, and combines proofs of equivalence [13].

The 'native' notion of equality in Coq, *Leibniz equality*, is that of terms being convertible, naturally reified as a proposition by the inductive type family eq with single constructor eq_refl : ∀ (T : Type)(x : T), x ≡ x, where a ≡ b is notation for eq T a b. Since convertibility is a congruence, a proof of a ≡ b lets us substitute b for a anywhere inside a term without further conditions. Our interest is in more complicated equalities, so we diverge from Coq tradition and reserve = for setoid equality. Rewriting with = *does* give rise to side conditions. For instance, consider formal fractions of integers as a representation of rationals. Rewriting a subterm using such an equality is permitted only if the subterm is an argument of a function that has been proven to *respect* the equality. Such a function is called Proper, and that property must be proved for each function in whose arguments we wish to enable rewriting.

Type classes. Type classes have been a great success story in the Haskell functional programming language, as a means of organizing interfaces of abstract structures. Coq's type classes provide a superset of their functionality, but are implemented in a different way.

In Haskell and Isabelle, type classes and their instances are second class. They are handled as specialized syntactic constructs whose semantics are given specifically by the type class apparatus. By contrast, the expressivity of dependent types and inductive families as supported in Coq, combined with the use of pre-existing technology in the system (namely proof search and implicit arguments) enable a *first class* type class implementation [14]: classes are ordinary record types ('dictionaries'), instances are ordinary constants of these record types (registered as *hints* with the proof search machinery), class constraints are ordinary implicit parameters, and instance resolution is achieved by augmenting the unification algorithm to invoke ordinary proof search for implicit arguments of class type. Thus, type classes in Coq are realized by relatively minor syntactic aids that bring together existing facilities of the theory and the system into a coherent idiom, rather than by introduction of a new category of qualitatively different definitions with their own dedicated semantics.

We use the algebraic hierarchy based on type classes and its abstract specification of $\mathbb{N}, \mathbb{Z}$ and $\mathbb{Q}$ [15]. Unfortunately, we should note that we have clearly met the efficiency problems connected to the current implementation of type classes in Coq. Luckily, these efficiency problems are limited to instance resolution which is only performed at compile time. Type classes have only a very minor effect on the computation time of type checked terms due to the absence of code inlining; see Section 7 for timings.

Virtual machine and machine integers. Coq includes a virtual machine [16], vm_compute, based on Ocaml's virtual machine to allow efficient evaluation. Both the abstract machine and its compilation scheme have been proved correct, in Coq, with respect to the weak reduction semantics. However, we still need to extend our trusted core to a bigger kernel, as the *implementation* has not been formally verified.

Machine integers were also added to the Coq system [17]. The usual evaluation inside Coq (compute) uses a special inductive type for cyclic integers, but the virtual machine uses Ocaml's machine integers. This allows for a big speed-up, for which we pay by having to trust (the virtual machine and) that Ocaml treats these integers correctly. The time difference between computation with Coq's int and Ocaml's Big_int is about a factor of 20 [18] on primality tests.

3 Metric spaces

Having completed our brief description of the Coq-system, we now turn to O'Connor's formalization of exact real numbers [2]. Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \to \mathbb{R}_+$ satisfying certain axioms. The usual constructive formulation requires d be a computable function. We use a more relaxed definition of a metric space that does not require the metric be a function. A similar construction can be found in the work by Richman [19]. The metric is represented via a (respectful) ball relation $\mathbf{B} : \mathbb{Q}_+ \to X \to X \to$ Prop, where Prop is the type of propositions, satisfying five axioms:

msp_refl : $\forall x\, \varepsilon,\ \mathbf{B}_\varepsilon\, x\, x$

$\mathsf{msp\_sym} : \forall x\,y\,\varepsilon,\ \mathbf{B}_\varepsilon\,x\ y \to \mathbf{B}_\varepsilon\,y\ x$
$\mathsf{msp\_triangle} : \forall x\,y\,z\,\varepsilon_1\,\varepsilon_2,\ \mathbf{B}_{\varepsilon_1}\,x\ y \to \mathbf{B}_{\varepsilon_2}\,y\ z \to \mathbf{B}_{\varepsilon_1+\varepsilon_2}\,x\ z$
$\mathsf{msp\_closed} : \forall x\,y\,\varepsilon,\ (\forall \delta,\ \mathbf{B}_{\varepsilon+\delta}\,x\ y) \to \mathbf{B}_\varepsilon\,x\ y$
$\mathsf{msp\_eq} : \forall x\,y,\ (\forall \varepsilon,\ \mathbf{B}_\varepsilon\,x\ y) \to x = y$

The ball relation $\mathbf{B}_\varepsilon\,x\ y$ expresses that the points x and y are within ε of each other. We call this a ball relationship because the partially applied relation $\mathbf{B}^X_\varepsilon\,x : X \to \mathsf{Prop}$ is a predicate that represents the closed ball of radius ε around the point x. For example, the ball relation on $\mathbb{Q}$ is $\mathbf{B}^{\mathbb{Q}}_\varepsilon\,x\ y := |x-y| \le \varepsilon$.

A metric space X is a *prelength* space if:

$$\forall a\,b\,\varepsilon\,\delta_1\,\delta_2,\ \varepsilon < \delta 1 + \delta 2 \to \mathbf{B}_\varepsilon\,a\ b \to \exists c,\ \mathbf{B}_{\delta_1}\,a\ c\ \wedge\ \mathbf{B}_{\delta_2}\,c\ b.$$

This property states that if two points a and b within ε of each other, then there exists curve of length $d(a,b)$ in the completion of X that connects a and b. The metric space $\mathbb{Q}$ is a prelength space.

We will introduce the completion of a prelength space as a monad. In order to do this we will first introduce monads.

Monads. Moggi [20] recognized that many non-standard forms of computation may be modeled by monads[1]. Wadler [21] popularized their use in functional programming. Monads are now an established tool to structure computation with side-effects. For instance, programs with input X and output Y which have access to a mutable state S can be modeled as functions of type $X \times S \to Y \times S$, or equivalently $X \to (Y \times S)^S$. The type constructor $\mathfrak{M}Y := (Y \times S)^S$ is an example of a monad. Similarly, partial functions may be modeled by maps $X \to Y_\bot$, where $Y_\bot := Y + ()$ is a monad.

The formal definition of a (strong) monad is a triple $(\mathfrak{M}, \mathsf{return}, \mathsf{bind})$ consisting of a type constructor $\mathfrak{M}$ and two functions:

$$\begin{aligned} \mathsf{return} &: X \to \mathfrak{M}X \\ \mathsf{bind} &: (X \to \mathfrak{M}Y) \to \mathfrak{M}X \to \mathfrak{M}Y \end{aligned}$$

We will denote $\mathsf{return}\,x$ as $\hat{x}$, and $\mathsf{bind}\,f$ as $\check{f}$. These two operations must satisfy the following laws:

$$\begin{aligned} \mathsf{bind}\ \ \mathsf{return}\ \ a &= a \\ \check{f}\,\hat{a} &= f\,a \\ \check{f}\,(\check{g}\,a) &= \mathsf{bind}\ (\check{f} \circ g)\,a \end{aligned}$$

Completion monad. Given a metric space X, the completion $\mathfrak{C}X$ of X is defined by:

$$\mathfrak{C}X := \{f : \mathbb{Q}_+ \to X \mid \forall \varepsilon_1\,\varepsilon_2,\ \mathbf{B}_{\varepsilon_1+\varepsilon_2}\,(f\,\varepsilon_1)\ (f\,\varepsilon_2)\}.$$

Given metric spaces X and Y, a function $f : X \to Y$ is *uniformly continuous* with *modulus* $\mu_f : Q_+ \to Q_+$ if:

$$\forall \varepsilon\,x_1\,x_2,\ \mathbf{B}_{\mu_f \varepsilon}\,x_1\ x_2 \to \mathbf{B}_\varepsilon\,(f\,x_1)\ (f\,x_2).$$

[1] In category theory one would speak about the Kleisli category of a (strong) monad.

Completion is a monad on the category of prelength spaces with uniformly continuous functions. The function return : $X \to \mathfrak{C}X$ defined by $\lambda x\, \varepsilon,\, x$ is the embedding of a prelength space in its completion. Moreover, a uniformly continuous function $f : X \to \mathfrak{C}Y$ with modulus μ_f can be lifted to operate on complete prelength spaces as bind $f : \mathfrak{C}X \to \mathfrak{C}Y$ defined by $\lambda x\, \varepsilon,\, f\,(x\,(\mu_f \frac{\varepsilon}{2}))\, \frac{\varepsilon}{2}$. The restriction to prelength spaces allows the present efficient definition of bind.

One advantage of this approach is that it helps us to work with simple representations. Let $\mathbb{R} := \mathfrak{C}\mathbb{Q}$. Then to specify a function from $\mathbb{R} \to \mathbb{R}$, we define a uniformly continuous function $f : \mathbb{Q} \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$ as the required function. Hence, the completion monad allows us to do in a structured way what was already folklore in constructive mathematics: to work with simple, often decidable, approximations to continuous objects.

4 Abstract interfaces using type classes

An important part of this work is to further develop the algebraic hierarchy based on type classes by Spitters and van der Weegen [15]. Especially, we have formalized some order theory and developed interfaces for mathematical operations common in programming languages such as shift and power. This layer of abstraction makes both proof engineering and programming more flexible: it avoids duplication of code, it introduces a canonical way to refer to operations and properties, both by names and notations, and it allows to easily swap different implementations of number representations and their operations. First we will briefly recap the design decisions made in [15].

Algebraic structures are expressed in terms of a number of carrier sets, a number of relations and operations, and a number of laws that the operations satisfy. One way of describing such a structure is by a *bundled representation*: one uses a dependently typed record that contains the carrier, operations and laws. For example a semigroup can be represented as follows. (The fields sg_car and sg_proper support our explicit handling of naive set theory in type theory.)

```
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) }
```

However, this approach has some serious limitations, the most important one being a lack of support for *sharing* components. For example, suppose we group together two CommutativeMonoids in order to create a SemiRing. Now awkward hacks are necessary to establish equality between the carriers. A second problem is that if we stack up these records to represent higher structures the projection paths become increasingly large.

Historically these problems have been an acceptable trade-off because *unbundled representations*, in which the carrier and operations are parameterized, introduce even more problems. Spitters and van der Weegen have proposed a use of Coq's new type class machinery that resolves many of the problems of unbundled representations. Our current experiment confirms that this is a viable approach.

An *operational type class* is defined for each operation and relation.

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

Now an algebraic structure is just a type class living in Prop that is parametrized by its carrier, relations and operations. This class contains all laws that the operations should satisfy. Since the operations are unbundled we can easily support sharing. For example let us consider the SemiRing interface.

```
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 }.
```

Without type classes it would be a burden to manually carry around the carrier, relations and operations. However, because these parameters are just type class instances, the type class machinery will perform that job for us. For example,

```
Lemma double `{SemiRing R} x : 2 ∗ x = x + x.
```

The backtick instructs Coq to automatically insert implicit declarations, namely e plus mult zero one. Furthermore, instance resolution will automatically find instances of the operational type classes for the written notations.

This approach to interfaces proved useful to formalize a standard algebraic hierarchy. Combined with category theory and universal algebra, $\mathbb{N}$ and $\mathbb{Z}$ are represented as interfaces specifying an initial SemiRing and initial Ring [15]. These abstract interfaces for the naturals an integers make it easier to change the concrete representation in the future. No such simple specification for $\mathbb{Q}$ seems to exists, so we choose to specify it as the field of fractions of $\mathbb{Z}$. More precisely, $\mathbb{Q}$ is specified as a Field containing $\mathbb{Z}$ that moreover can be embedded into the field of fractions of $\mathbb{Z}$.

```
Inductive Frac R `{e : Equiv R} `{zero : RingZero R} : Type :=
  frac { num : R ; den : R ; den_nonzero : den ≠ 0 }.
Class RationalsToFrac (A : Type) := rationals_to_frac : ∀ B `{Integers B}, A → Frac B.
Class Rationals A {e plus mult zero one opp inv} `{U : !RationalsToFrac A} : Prop := {
  rationals_field :> @Field A e plus mult zero one opp inv ;
  rationals_frac :> ∀ `{Integers Z}, Injective (rationals_to_frac A Z) ;
  rationals_frac_mor :> ∀ `{Integers Z}, SemiRing_Morphism (rationals_to_frac A Z) ;
  rationals_embed_ints :> ∀ `{Integers Z}, Injective (integers_to_ring Z A) }.
```

4.1 Order theory

To abstract from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ and their various implementations, we provide a basic library for order theory and its interaction with algebraic structures. For example,

```
Class RingOrder ‘{Equiv A} ‘{RingPlus A} ‘{RingMult A} ‘{RingZero A}
    (o : Order A) := {
  ringorder_partialorder :> PartialOrder (≤) ;
  ringorder_plus :> ‘(OrderPreserving (z +));
  ringorder_mult : ‘(0 ≤ x → ∀ y, 0 ≤ y → 0 ≤ x ∗ y) }.
```

To apply this to $\mathbb{N}$, which is merely a semiring, we introduce the, apparently new, notion of a SemiRingOrder. Every RingOrder is a SemiRingOrder.

```
Class SemiRingOrder ‘{Equiv A} ‘{RingPlus A} ‘{RingMult A} ‘{RingZero A}
    (o : Order A) := {
  srorder_partialorder :> PartialOrder (≤) ;
  srorder_plus : ‘(x ≤ y ↔ ∃ z, 0 ≤ z ∧ y = x + z) ;
  srorder_mult : ‘(0 ≤ x → ∀ y, 0 ≤ y → 0 ≤ x ∗ y) }.
```

This allows us to refer by canonical names to lemmas as those shown below for $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and the dyadics.

```
Lemma plus_compat x1 y1 x2 y2 : x1 ≤ y1 → x2 ≤ y2 → x1 + x2 ≤ y1 + y2 .
Lemma sprecedes_1_2 : 1 < 2.
```

For instances of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ it is easy to define an order satisfying these interfaces:

```
Instance nat_precedes ‘{Naturals N} : Order N | 10 := λ x y, ∃ z, y = x + z.
```

However, often we encounter an a priori different order on a structure, most likely an order defined in Coq's standard library (like Nle on N). Therefore we prove that an arbitrary order satisfying these interfaces while also being a TotalOrder uniquely specifies the order on $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$. For example:

```
Context ‘{Naturals N} ‘{Naturals N2} {f : N → N2} ‘{!SemiRing_Morphism f}
  {o1 : Order N} ‘{!SemiRingOrder o1} ‘{!TotalOrder o1}
  {o2 : Order N2} ‘{!SemiRingOrder o2} ‘{!TotalOrder o2}.
Global Instance: OrderEmbedding f.
```

4.2 Basic operations

The operation nat_pow is most commonly defined as repeated multiplication and the operation shiftl is defined as repeated multiplication by 2. However, if we implement these operations that way, they become too slow for the purpose of our work: efficient computation with real numbers. Instead we specify the desired behavior of these operations. This approach allows for different implementations for different number representations and avoids definitions and proofs becoming implementation dependent.

We introduce interfaces that specify the behavior of the operations abs, shiftl, shiftr, nat_pow, int_pow and Euclidean division. Again there are various ways of specifying these interfaces: with Σ-types, bundled or unbundled. In general, Σ-types are convenient for functions whose specification is easy, for example:

```
Class Abs A ‘{Equiv A} ‘{Order A} ‘{RingZero A} ‘{GroupInv A}
    := abs_sig: ∀ (x : A), { y : A | (0 ≤ x → y = x) ∧ (x ≤ 0 → y = −x)}.
Definition abs ‘{Abs A} := λ x : A, ‘ (abs_sig x).
```

However, for more complex operations, such as shiftl, such an interface is different from the usual mathematical specification because we cannot quantify over all possible input values. Now there are two ways: a bundled or an unbundled interface. Since these interfaces are not used for hierarchies the disadvantages of the latter do not apply. Let us first describe the former approach.

```
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).
```

Although this interface seems reasonable, it does not work well in Coq. The simpl tactic which is used to simplify a goal will unfold occurrences of shiftl to their underlying definition (for example in case of BigN, the expression x ≪ n becomes BigN.shiftl x n). This is rather inconvenient because Coq will then be unable to use lemmas concerning ≪ for rewriting. This problem is caused because shiftl is a projection of a record, which is in fact a δ-redex that will be unfolded by simpl. Currently there seems to be no way to adjust the behavior of simpl to remove this inconvenience. A similar problem was already observed in SSReflect [22].

Instead we use an unbundled interface, which has a lot in common with our interfaces for algebraic structures. Now shiftl no longer contains a δ-redex.

```
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 }.
```

We do not specify shiftl as shiftl x n = x ∗ 2 ˆ n since on the dyadics we cannot take a negative power while we can shift by a negative integer.

4.3 Decision procedures

The Decision type class collects types with a decidable equality [15].

```
Class Decision P := decide: sumbool P (¬ P).
```

Using this type class we can declare a parameter ‘{∀ x y, Decision (x ≤ y)} to describe a decider for ≤ and say decide (x ≤ y) to decide whether x ≤ y or not. This type class allows to easily define additional deciders, like the one for the strict order. We have to be careful however. Consider the order on the dyadics.

```
Global Instance dy_equiv: Equiv Dyadic := λ x y,
 ZtoQ (mant x) ∗ 2 ˆ (expo x) ≤ ZtoQ (mant y) ∗ 2 ˆ (expo y)
```

Now, decide (x ≤ y) is actually @decide Dyadic (x ≤ y) dyadic_dec, where dyadic_dec is the computational conclusion of the decision. Due to eager evaluation, and the absence of dead code removal, the second argument, $x \le y$, is also evaluated. Evaluation of this argument results in a conversion of x and y into Q, as described above. But since this argument is just a proposition it is later thrown away. We avoid this problem introducing a λ-abstraction.

```
Definition decide_rel `(R : relation A) {dec : ∀ x y, Decision (R x y)}
   (x y : A) : Decision (R x y) := dec x y.
```

We can now define:

```
Context `{!PartialOrder (≤) } {!TotalOrder (≤) } `{∀ x y, Decision (x ≤ y)}.
Global Program Instance sprecedes_dec: ∀ x y, Decision (x < y) | 9 := λ x y,
  match decide_rel (≤) y x with
  | left E ⇒ right _
  | right E ⇒ left _
  end.
```

4.4 Approximate rationals

In order to make our implementation of the reals independent of the underlying dense set, we provide an abstract specification of such a set. The interface of this specification is called *approximate rationals* and is inspired by the notion of *approximate fields* which is used in the HASKELL implementation of the exact reals by Bauer and Kavler [23]. We provide an implementation of this interface by dyadics based on COQ's machine integers.

Our interface describes an ordered ring containing Z that is dense in Q. Here Z are the binary integers from COQ's standard library, and Q are the rationals based on these binary integers. We do not parametrize by arbitrary integer and rational implementations because they are hardly used for computation.

Also, for efficient computation, this interface contains the operations: approximate division, normalization, an embedding of Z, absolute value, power by N, shift by Z, and decision procedures for both equality and order.

```
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 }.
```

O'Connor's [2] keeps the size of the rational numbers small to avoid efficiency problems. He introduced a function approx x ϵ that yields the 'simplest' rational number between x $-$ ϵ and x $+$ ϵ. In our interface we modify the approx function slightly: app_approx x k yields an arbitrary element between x $- 2^k$ and x $+2^k$. Using this function we define the compress operation on the real numbers: compress := bind (λ ϵ, app_approx x (Qdlog2 ϵ)) such that compress x = x.

In Section 5 we will explain our choice of using a power of 2 to specify the precision of app_div and app_approx. In the remainder of this section we briefly describe our implementation by the dyadics.

The dyadic rationals are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$. In order to remain independent of an integers implementation, we abstract over it. For our eventual implementation of the approximate rationals we use Coq's machine integers, bigZ. Now given an arbitrary integer implementation Int it is straightforward to define the dyadics. Here we will just show the ring operations.

```
Notation "x ↾ p" := (exist _ x p) (at level 20).
Record Dyadic := dyadic { mant : Int ; expo : Int }.
Infix "$" := dyadic (at level 80).
Global Instance dy_inject: Coerce Int Dyadic := λ x, x $ 0.
Global Instance dy_opp: GroupInv Dyadic := λ x, −mant x $ expo x.
Global Instance dy_mult: RingMult Dyadic := λ x y, mant x ∗ mant y $ expo x + expo y.
Global Instance dy_0: RingZero Dyadic := ('0:Dyadic).
Global Instance dy_1: RingOne Dyadic := ('1:Dyadic).
Global Program Instance dy_plus: RingPlus Dyadic := λ x y,
  if decide_rel (≤) (expo x) (expo y)
  then mant x + mant y ≪ (expo y − expo x) ↾ _ $ min (expo x) (expo y)
  else mant x ≪ (expo x − expo y) ↾ _ + mant y $ min (expo x) (expo y).
```

In this code shiftl has type Int $\rightarrow$ Int$^+$$\rightarrow$ Int, where Int$^+$ is a Σ-type describing the non-negative elements of Int. Therefore, in the definition of dy_plus we have to equip expo y $-$ expo x with a proof that it is in fact non-negative.

5 Power series

Elementary transcendental functions as exp, sin, ln and arctan can be defined by their power series. A power series is particularly suited for computation if its coefficients are alternating, decreasing and have limit 0. For $-1 \leq x \leq 0$,

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

is of this form. To approximate exp x with error ε we take the partial sum until $\frac{x^i}{i!} \leq \varepsilon$. In order to implement this efficiently we use a stream representing the series and define a function that sums the required number of elements. For example, the series 1, a, a^2, a^3, ... is defined by the following stream.

```
CoFixpoint powers_help (c : A) : Stream A := Cons c (powers_help (c ∗ a)).
Definition powers : Stream A := powers_help 1.
```

Streams in Coq, like lists in Haskell, are lazy. So, in the example the multiplications are accumulated.

Since Coq only allows structural recursion it requires some work to convince Coq that our algorithm terminates. Intuitively, one would describe the limit as an upperbound of the required number of elements using the Exists predicate.

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

This approach leads to performance problems. The upperbound, encoded in unary format, may become very large while generally only a few terms are necessary. Due to vm_compute's eager evaluation scheme, this unary number will be computed before summing the series. Instead we use LazyExists [24].

```
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.
```

O'Connor's InfiniteAlternatingSum s returns the real number represented by the infinite alternating sum over s, where the stream s is decreasing, non-negative and has limit 0. We have extended this in two ways. First, by generalizing some of the work to abstract structures. Second, as we do not have exact division on approximate rationals, we extended his algorithm to work with approximate division. The latter required changing InfiniteAlternatingSum s to InfiniteAlternatingSum n d which computes the infinite alternating sum of the stream $\lambda i, \frac{n_i}{d_i}$. This allows us to postpone divisions. Also, we have to determine both the length of the partial sum and the required precision of the divisions. To do so we find k such that:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app\_div}\ n_k\ d_k\ (\mathsf{log}\frac{\varepsilon}{2k}) + \frac{\varepsilon}{2k})\ 0. \tag{1}$$

Now k is the length of the partial sum, and $\frac{\varepsilon}{2k}$ is the required precision of division. Using O'Connor's results we have verified that these values are correct and such a k indeed exists for a decreasing, non-negative stream with limit 0.

As noted in Section 4.4, we have specified the precision of division in powers of 2 instead of using a rational value. This is allows us to replace (1) with:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app\_div}\ n_k\ d_k\ (\mathsf{log}\ \varepsilon - (k+1)) + 1 \ll (\mathsf{log}\ \varepsilon - (k+1)))\ 0.$$

Here k is the length of the partial sum, and 2^l, where $l = \mathsf{log}\ \varepsilon - (k+1)$, is the required precision of division. This variant can be implemented without any arithmetic on the rationals and is thus much more efficient.

This method gives us a fast way to compute the infinite alternating sum, in practice, only few extra terms have to be computed and due to the approximate division the auxiliary results are kept as small as possible.

Using this method to compute infinite alternating sums we have so far implemented exp and arctan. Furthermore, we extend the exponential to its complete domain by repeatedly applying the following formula.

$$\mathsf{exp}\ x = (\mathsf{exp}(x \ll 1))^2 \tag{2}$$

Our tests have shown that reducing the input to a value between $-2^k \leq x \leq 0$ for $50 \leq k$ yields major performance improvements as the series will converge much faster. For higher precisions setting it to $75 \leq k$ gives even better results.

By defining arctan on $[0, 1)$, we can define

$$\pi := 176 * \mathsf{arctan}\frac{1}{57} + 28 * \mathsf{arctan}\frac{1}{239} - 48 * \mathsf{arctan}\frac{1}{682} + 96 * \mathsf{arctan}\frac{1}{12943}.$$

Since we do not have exact devision on the approximate rationals, we here see the purpose of parameterizing infinite sums by two streams.

6 Square root

We use Wolfram's algorithm [25, p.913] for computing the square root. Its complexity is linear, in fact it provides a new binary digit in each step. We aim to investigate Newton iteration in future work.

```
Context `(Pa : 1 ≤ a ≤ 4).
Fixpoint AQroot_loop (n : nat) : AQ * AQ :=
  match n with
  | O ⇒ (a, 0)
  | S n ⇒
      let (r, s) := AQroot_loop n in
      if decide_rel (≤) (s + 1) r
      then ((r − (s + 1)) ≪ (2:Z), (s + 2) ≪ (1:Z))
      else (r ≪ (2:Z), s ≪ (1:Z))
  end.
```

Three easy invariants allow us to prove this series converges to the square root.

```
Lemma AQroot_loop_invariant1 (n : nat) :
 snd (AQroot_loop n) * snd (AQroot_loop n) + 4 * fst (AQroot_loop n) = 4 * 4 ˆ n * a.
Lemma AQroot_loop_invariant2 (n : nat) :
 fst (AQroot_loop n) ≤ 2 * snd (AQroot_loop n) + 4.
Lemma AQroot_loop_fst_bound (n : nat) :
 fst (AQroot_loop n) ≤ 2 ˆ (3 + n).
```

7 Benchmarks

The first step in this research was to create a Haskell prototype based on O'Connor's implementation of the real numbers in Haskell [2]. The second step was to implement this prototype in Coq. Currently, our Coq development contains the field operations, computation of power series, exp, arctan, π and the square root. Apart from the square root, the correctness of these operations has been verified in the Coq system.

In this section we present some benchmarks comparing the old and the new implementation, both in Haskell and Coq. All benchmarks have been carried out on an Intel Core Quad 2.4 GHz with 8GB of memory running

Expression	Decimals	O'Connor	Krebbers/Spitters
$\mathsf{sin}\ (\mathsf{sin}\ (\mathsf{sin}\ 1))$	10,000	71s	5s
$\mathsf{cos}\ (10^{50})$	10,000	2.7s	0.6s
$\tan\ (\sqrt{2}) + \mathsf{arctanh}\ (\mathsf{sin}\ 1)$	500	133s	2.2s

Table 1. Haskell, compiled with `ghc` version 6.12.1, using `-O2`.

Expression	Decimals	O'Connor	Krebbers/Spitters
π	300	55s	0.8s
$\mathsf{exp}\ (\mathsf{exp}\ (\mathsf{exp}\ (\frac{1}{2})))$	25	123s	0.23s
$\mathsf{exp}\ \pi - \pi$	25	52s	0.1s
$\mathsf{arctan}\ \pi$	25	134s	1.0s

Table 2. Coq trunk revision 13841.

Debian GNU/Linux with kernel 2.6.32. The sources of our developments can be found at `http://robbertkrebbers.nl/research/reals`.

Table 1 shows some benchmarks in Haskell with compiler optimizations enabled (`-O2`) and Table 2 compares our Coq implementation with O'Connor's. More extensive benchmarking shows that our Haskell implementation generally benefits from a 15 times speed up while the speed up in Coq is usually more than a 100 times. This difference is explained by the fact that O'Connor's Haskell implementation already used fast integers, while his Coq implementation did not. In the same times as shown in Table 2 for the old implementation, the new implementation is able to compute the first 2,000 decimals of π, 450 decimals of $\mathsf{exp}\ (\mathsf{exp}\ (\mathsf{exp}\ (\frac{1}{2})))$, 425 decimals of $\mathsf{exp}\ \pi - \pi$ and 85 decimals of $\mathsf{arctan}\ \pi$. This is an improvement of up to 18 times of the number of decimals.

It is interesting to notice that π and arctan benefit the least from our improvements, as we are unaware of an optimization similar to the squaring trick for exp (Section 5, Equation 2).

We conclude this section with a comparison between the performance of Wolfram's algorithm in Coq and Haskell. The Haskell prototype (without compiler optimizations) is quite fast, computing 10,000 iterations (giving 3,010 decimals) of $\sqrt{2}$ takes 0.2s. In Coq it takes 11.6s using type classes and 11.3s without type classes. Here we exclude the time spend on type class resolution. Thus type classes cause only a 3% performance penalty on computations.

Unfortunately, the Coq-implementation is slow compared to Haskell. Laurent Théry suggested that this is due to the representation of the fast integers, which uses a tree with a fixed depth and when the size of the integer becomes too big uses a less optimal representation. Increasing the size of the tree representation and avoiding an inefficiency in the implementation of shifts reduces this time to 7.5s.

8 Conclusions and Related work

We have greatly improved the performance of real number computation in Coq using Coq's new machine integers. We produced highly structured and abstract code using type classes with no apparent performance penalty. Moreover, Coq's notation mechanism combined with unicode characters gives nicely readable statements and proofs. Type classes were a great help in our work. However, the current implementation of instance resolution is still experimental and at times too slow (at compile time). The use of canonical structures by the Ssreflect team does not suffer from these efficiency issues [26]. However, their library is currently not suited for setoids which are crucial to us. We hope these issues might be addressed by the integration of unification hints [27] into Coq.

We needed to adapt our correctness proofs to prevent the virtual machine from eagerly evaluating them. Lazy evaluation for Prop would have allowed us to use the original proofs.

The experimental native_compute performs evaluation by compilation to native Ocaml code. This approach uses the Ocaml compiler available and is interesting for heavy compilation. Our first experiments indicate a 10 times speed up with Wolfram iteration. Unfortunately, native_compute does not work with Coq trunk yet, so we were unable to test it with our implementation of the reals.

The Flocq project [28] formalizes floating-points in Coq. It provides a library of theorems on a multi-radix multi-precision arithmetic and supports efficient numerical computations inside Coq. However, the current library is still too limited for our purposes, but in the future it should be possible to show that they form an instance of our approximate rationals. This may allow us to gain some speed by taking advantage of fine grained algorithms on the floats instead of our more straightforward ones.

The encoding of real numbers as streams of 'bits' is potentially interesting. However, currently there is a big difference in performance. The computation of 37 decimals of the square root of 1/2 by Newton iteration [29] took 12s. This should be compared with our use of the Wolfram iteration, which gives only linear convergence, but with which we nevertheless obtain 3,000 decimals in in a similar time.

The present work is part of a larger program to use constructive mathematics based on type theory as a programming language for exact analysis. This should culminate in a numerical ODE-solver.

9 Acknowledgements

We thank Eelis van der Weegen for many discussions. We thank Pierre Letouzey and Matthieu Sozeau for closing some of our bug reports.

References

1. Bishop, E.A.: Foundations of constructive analysis. McGraw-Hill (1967)

2. O'Connor, R.: A Monadic, Functional Implementation of Real Numbers. MSCS **17**(1) (2007) 129–159
3. O'Connor, R.: Certified Exact Transcendental Real Number Computation in Coq. In: TPHOLs 2008. Volume 5170 of LNCS. (2008) 246–261
4. O'Connor, R., Spitters, B.: A computer verified, monadic, functional implementation of the integral. TCS **411**(37) (2010) 3386–3402
5. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation **76**(2-3) (1988) 95–120
6. Coquand, T., Paulin, C.: Inductively defined types. In: COLOG-88. Volume 417 of LNCS. Springer (1990) 50–66
7. Coq Development Team: The Coq Proof Assistant Reference Manual. INRIA-Rocquencourt (2008)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in TCS. Springer (2004)
9. Martin-Löf, P.: An intuitionistic theory of types. In: Twenty-five years of constructive type theory. Volume 36 of Oxford Logic Guides. OUP (1998) 127–172
10. Martin-Löf, P.: Constructive Mathematics and Computer Science. In: Logic, Methodology and the Philosophy of Science VI. Volume 104 of Studies in Logic and the Foundations of Mathematics. (1982) 153–175
11. Hofmann, M.: Extensional constructs in intensional type theory. CPHC/BCS Distinguished Dissertations. Springer (1997)
12. Palmgren, E.: Constructivist and Structuralist Foundations: Bishop's and Lawvere's Theories of Sets. Technical Report 4, Mittag-Leffler (2009)
13. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning **2**(1) (2009) 41–62
14. Sozeau, M., Oury, N.: First-class type classes. In: TPHOLs 2008. Volume 5170 of LNCS. (2008) 278–293
15. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. MSCS, special issue on "Interactive theorem proving and the formalization of mathematics" (2011)
16. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP. (2002) 235–246
17. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: ITP 2010. Volume 6172 of LNCS. (2010) 83–98
18. Spiwack, A.: Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory. PhD thesis, INRIA (2011)
19. Richman, F.: Real numbers and other completions. Mathematical Logic Quarterly **54**(1) (2008) 98–108
20. Moggi, E.: Computational lambda-calculus and monads. In: LICS. (1989) 14–23
21. Wadler, P.: Monads for functional programming. In: Proceedings of the Marktoberdorf Summer School on Program Design Calculi. (August 1992)
22. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA (2008)
23. Bauer, A., Kavkler, I.: A constructive theory of continuous domains suitable for implementation. Annals of Pure and Applied Logic **159**(3) (2009) 251–267
24. O'Connor, R.: Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory. PhD thesis, Radboud University Nijmegen (2009)
25. Wolfram, S.: A new kind of science. Wolfram Media (2002)
26. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 327–342

27. Asperti, A., Ricciotti, W., Coen, C., Tassi, E.: Hints in Unification. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 84–98
28. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: Proc 20th IEEE Symposium on Computer Arithmetic. (2011)
29. Julien, N., Pasca, I.: Formal Verification of Exact Computations Using Newton's Method. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 408–423