TYPE CLASSES FOR EFFICIENT EXACT
REAL ARITHMETIC IN COQ

ROBBERT KREBBERS AND BAS SPITTERS

Institute for Computing and Information Sciences, Radboud University Nijmegen
e-mail address: mail@robbertkrebbers.nl, spitters@cs.ru.nl

ABSTRACT. Floating point operations are fast, but require continuous effort by the user to ensure correctness. This burden can be shifted to the machine by providing a library of *exact* analysis in which the computer handles the error estimates. Previously, we provided a fast implementation of the exact real numbers in the Coq proof assistant. This implementation incorporates various optimizations to speed up the basic operations of O'Connor's implementation by a 100 times. We implemented these optimizations in a modular way using type classes to define an abstract specification of the underlying dense set from which the real numbers are built. This abstraction does not hurt the efficiency.

This article is a substantially expanded version of (Krebbers/Spitters, Calculemus 2011) in which the implementation is extended in the various ways. First, we implement and verify the sine and cosine function. Secondly, we create an additional implementation of the dense set based on Coq's fast rational numbers. Thirdly, we extend the hierarchy to capture order on undecidable structures, while it was limited to decidable structures before. This hierarchy, based on type classes, allows us to share theory on the naturals, integers, rationals, dyadics, and reals in a convenient way. Finally, we obtain another dramatic speed-up by avoiding evaluation of termination proofs at runtime.

1. INTRODUCTION

There is a big gap between numerical algorithms in research papers, which typically use concepts like Hilbert spaces and fixed point theorems from functional analysis, and their actual implementation, which uses floating point numbers[1] and matrix operations. This gap makes it difficult to trust the code. Similarly, this gap is undesirable in proofs of theorems (e.g. Kepler conjecture [Hal02], existence of the Lorentz attractor [Tuc02]) that rely on the execution of this code for numerical approximation. Finally, from a purely software

[1]By floating points we mean numbers of the shape $n * b^e$, where n and e are bounded integers and b is the base for scaling (typically 2, 10 or 16). The most widely used form of floating point arithmetic is the IEEE 754 standard, which is present in many hardware and software implementations.

engineering point of view, the situation is undesirable, because the gap between the (abstract mathematical) numerical algorithms and the (concrete floating point) implemented program makes the code difficult to maintain.

The challenge to close this gap has already been posed by Bishop in his fundamental work on constructive analysis [Bis67]. Bishop proposed to use constructive analysis to bridge this gap. Moreover, we can narrow this gap by using

- exact real numbers or intervals instead of floating point numbers;
- functional programming instead of imperative programming;
- dependent type theory which allows us to compute with complete metric spaces;
- a proof assistant which allows us to verify the correctness proofs;
- constructive mathematics to tightly connect mathematics with computations and to avoid computationally impossible case distinctions.

Separately, all these tools have proved itself. By going to the limits of this proven technology we should be able to come within a small constant factor of floating point computations. In this way one would obtain a tool suitable for research and education in numerical analysis that allows one to compute abstractly at the level of functional analysis, e.g. to compute fixed points of operators on Hilbert spaces. Like the development of FORTRAN and MATLAB this will require a huge amount of work. In the present paper we focus on the performance of real number computation in the COQ proof assistant.

Real numbers, being infinite objects, cannot be represented exactly in a computer. Hence, in constructive analysis [Bis67] one uses functions which when fed a desired precision approximate a real numbers by a rational, or a dyadic number, to within that precision[2]. The real numbers are the completion of the rationals. This completion construction can be organized in a monad, a familiar construct from functional programming. The completion monad provides an efficient combination of proving and computing [O'C07]. In this way, O'Connor [O'C08] implements exact real numbers and the transcendental functions on them in COQ. A number of possible improvements in this implementation were already suggested in [OS10, O'C09].

(1) Use COQ's new machine integers instead of the integers built from ordinary inductive data types;
(2) use dyadic rationals (that are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$, also known as infinitary floats) instead of ordinary rationals;
(3) use approximate division to improve the implementation of power series.

Here we carry out all three optimizations. Unfortunately, changing O'Connor's implementation to use the new machine integers was far from trivial, as he used a particular concrete representation of the rationals. To avoid this in the future, we moreover provide an abstract specification of the dense set as *approximate rationals*. Finally, we obtain another dramatic speed-up by avoiding evaluation of termination proofs at runtime.

Outline. Section 2 describes some aspects of the COQ proof assistant relevant for our development. Section 3 describes metric spaces, monads, and O'Connor's implementation of the real numbers [O'C07]. Section 4 extends Spitters and van der Weegen's approach to abstract interfaces using type classes [SvdW11]. Section 5 describes the theory of approximate rationals, our implementation of the real numbers, and deals with computing power series and

---

[2]One could argue that we capture only the definable, or computable, real numbers in this way. These issues are important and well-studied, see for instance [TvD88], but we will not go into them here.

the square root. We finish with some benchmarks in Section 6 and conclusions in Section 7. The sources of our developments can be found at `https://github.com/c-corn/corn`.

2. THE COQ-SYSTEM

The COQ proof assistant is based on the calculus of inductive constructions [CH88, CP90], a dependent type theory with (co)inductive types; see [Coq12, BC04]. 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.

2.1. **Notations.** COQ has an extensible mechanism for defining complex notations. We use this mechanism heavily, together with unicode symbols, to obtain notations that are closer to common mathematical practice. However, due to conflicts with standard COQ syntax, there are some small deviations. For example, we write $\forall x, Px$ instead of $\forall x.Px$. In this paper we tried to stay as close as possible to the notations in our COQ development.

2.2. **Types and propositions.** Propositions in COQ are types [ML98, ML82], 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 [Let08] is still very hard to use for programs with the complexity, in terms of depth of definitions, that we are interested in [CFS03, CFL06].

2.3. **Constructive indefinite description.** In spite of the restriction on Prop discussed in the previous paragraph, COQ allows recursive functions to use a value of Prop type to ensure termination [BC04, 14.2.3, 15.4]. In particular, this is used to prove *constructive indefinite description*, which states that given a decidable predicate over the natural numbers, a Prop based existential can be converted into a Type based one. Its formal statement can be found in the standard library:

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

Here the notation $\{x : A \mid P\ x\}$ for $P : A \to$ Prop denotes a Σ-type. This lemma can be seen as a form of Markov's principle in COQ. The algorithm does a bounded search for a new witness satisfying the predicate. The witness from the Prop based existential is only used to prove termination of the search. No information flows from the Prop universe to the Type universe because the witness found for the Type based existential is independent of the witness from the Prop based one.

2.4. **Equality, setoids, and rewriting.** Because the Coq type theory lacks quotient types (as it endangers the decidability of type checking), one usually bases abstract structures on a *setoid* ('Bishop set'): a type equipped with an equivalence relation [Bis67, Hof97]. This leads to a naive set theory as described by Palmgren [Pal09]. When the user attempts to substitute a given (sub)term using an equality, the system keeps track of, resolves, and combines proofs of equivalence [Soz09].

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 : $\forall$ (T : Type)(x : T), $x \equiv x$, where $a \equiv b$ is notation for eq T a b. Since convertibility is a congruence, a proof of $a \equiv b$ 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.

2.5. **Type classes.** Type classes are a great success 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 [SO08]: 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 arguments, 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 $\mathbb{Q}$ described in [SvdW11]. 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 effect the computation time of type checked terms due to the absence of code inlining. In an illustrative example the use of type classes caused only a 3% performance penalty; see Section 6.

2.6. **Virtual machine and machine integers.** Coq includes a virtual machine [GL02], 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 [AGST10]. The usual evaluation inside COQ (compute) uses a special inductive type for cyclic integers, but the virtual machine (vm_compute) uses actual machine integers. The type bigZ of arbitrary precision integers is built from binary trees of these cyclic integers. Primality tests in [Spi11] show a big speed-up compared to the inductively defined integers. Our work confirms this big speed-up gained by using machine integers. We pay for this speed-up, however, by having to trust the virtual machine and its translation to actual machine integers.

3. METRIC SPACES AND THE COMPLETION MONAD

Having completed our brief description of the COQ-system, we now turn to O'Connor's formalization of exact real numbers [O'C07]. Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \to \mathbb{R}_{\geq 0}$ satisfying certain axioms. We use a more relaxed definition of a metric space that does not require the metric be a function; see also [Ric08]. The metric is represented via a (respectful) ball relation $\mathbf{B} : \mathbb{Q}_{>0} \to X \to X \to$ Prop satisfying:[3]

msp_refl : $\forall x\, \varepsilon,\ \mathbf{B}_\varepsilon\, x\, x$
msp_sym : $\forall x\, y\, \varepsilon,\ \mathbf{B}_\varepsilon\, x\, y \to \mathbf{B}_\varepsilon\, y\, x$
msp_triangle : $\forall x\, y\, z\, \varepsilon_1\, \varepsilon_2,\ \mathbf{B}_{\varepsilon_1}\, x\, y \to \mathbf{B}_{\varepsilon_2}\, y\, z \to \mathbf{B}_{\varepsilon_1 + \varepsilon_2}\, x\, z$
msp_closed : $\forall x\, y\, \varepsilon,\ (\forall \delta,\ \mathbf{B}_{\varepsilon + \delta}\, x\, y) \to \mathbf{B}_\varepsilon\, x\, y$
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 $\mathbf{B}_\varepsilon^X\, x : X \to$ 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 $\mathbf{B}_\varepsilon^{\mathbb{Q}}\, x\, y := |x - y| \leq \varepsilon$.

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

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

In particular, a prelength space has approximate midpoints: for any $\delta > 0$ we can take $\delta_1 = \delta_2 = \varepsilon/2 + \delta$. Every complete prelength metric space is a length metric space. The metric space $\mathbb{Q}$ is a prelength space; see [O'C07] for details.

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

3.1. **Monads.** Moggi [Mog89] recognized that many non-standard forms of computation may be modeled by monads[4]. Wadler [Wad92] 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 $X \to Y_\perp$, where $Y_\perp := Y + ()$ is a monad.

---

[3]We use the positive rational numbers $\mathbb{Q}_{>0}$ instead of the non-negative relation numbers $\mathbb{Q}_{\geq 0}$ as it can be expressed that two points x and y are within 0 of each other by $\forall \varepsilon,\ \mathbf{B}_\varepsilon\, x\, y$.

[4]In category theory one would speak about the Kleisli category of a (strong) 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:

$$\mathsf{return} : X \to \mathfrak{M}X$$
$$\mathsf{bind} : (X \to \mathfrak{M}Y) \to \mathfrak{M}X \to \mathfrak{M}Y$$

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

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

3.2. **Completion monad.** The completion of a metric space X is defined by:

$$\mathfrak{C}X := \{f : \mathbb{Q}_{>0} \to X \mid \forall \varepsilon_1\,\varepsilon_2,\ \mathbf{B}_{\varepsilon_1+\varepsilon_2}\,(f\,\varepsilon_1)\,(f\,\varepsilon_2)\}.$$

Given metric spaces X and Y, a function $f : X \to Y$ is *uniformly continuous* with *modulus* $\mu_f : \mathbb{Q}_{>0} \to \mathbb{Q}_{>0}$ if:

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

Completion is a monad on the category of metric spaces with uniformly continuous functions. The function $\mathsf{return} : X \to \mathfrak{C}X$ defined by $\lambda x\,\varepsilon,\ x$ is the embedding of a metric 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 metric spaces as $\mathsf{bind}\, f : \mathfrak{C}X \to \mathfrak{C}Y$ defined by $\lambda x\,\varepsilon,\ f\,(x\,(\mu_f \frac{\varepsilon}{2}))\,\frac{\varepsilon}{2}$. A restriction to prelength spaces is essential for this efficient definition of bind; see [O'C07] for details.

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 [SvdW11]. Especially, we extend their hierarchy with constructive fields, order theory and interfaces for mathematical operations, such as shift and power, common in programming languages. 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 us to easily swap different implementations of number representations and their operations. First we will briefly recap the design decisions made in [SvdW11]. For a nice tutorial following this design see [CS12]. More information on type classes and setoids in Coq can also be found in the reference manual [Coq12].

Algebraic structures are expressed in terms of a number of carrier sets, a number of relations and operations, and a number of laws that these operations and relations must satisfy. One way of describing such a structure is by a *bundled representation* (as used

in [GPWZ02] for example): one uses a dependently typed record that contains the carrier, operations and laws. A setoid can be represented as follows.

```
Record Setoid : Type := {
   st_car :> Type;
   st_equiv : st_car → st_car → Prop;
   st_setoid : Equivalence st_eq }.
Infix "=" := st_equiv : type_scope.
```

The notation :> registers the projection st_car : Setoid → Type as a coercion. Using the above interface for Setoids, one can now define a SemiGroup whose carrier is a setoid.

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

The field sg_proper states that the operation sg_op respects the setoid equality. Its definition expands to $\forall x_1 \, x_2 \, , \, x_1 = x_2 \rightarrow \forall y_1 \, y_2 \, , \, y_1 = y_2 \rightarrow$ sg_op $x_1 \, y_1 =$ sg_op $x_2 \, y_2$.

However, this approach has some serious limitations, the most important one being a lack of support for *sharing* components. For example, suppose we want to group together two CommutativeMonoids in order to create a SemiRing. Now awkward hacks are necessary to establish equality between the carriers[5]. A second problem is that if we stack up these records to represent higher structures the projection paths become increasingly long. In the above example, the projection path to obtain the carrier of a semigroup G is st_car (sg_car G), but for fields, this path will be much longer.

Historically these problems have been an acceptable trade-off because *unbundled representations*, in which the carrier and operations are parameterized, introduce even more problems. An unbundled representation of a semigroup is as follows.

```
Record SemiGroup A (Ae : A → A → Prop) (Aop : A → A → A) : Prop := {
   sg_setoid : Equivalence Ae;
   sg_op_proper : Proper (Ae ⟹ Ae ⟹ Ae) Aop;
   sg_ass : ∀ x y z, Ae (Aop x (Aop y z)) (Aop (Aop x y) z) }
```

There is nothing to bind notation to, no structure inference, and declaring and passing requires too much manual bookkeeping. 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 alternative solution is provided by packed classes [GGMR09] which use an alternative, and older, implementation of a variant of type classes: canonical structures; see also Section 7. Yet another approach would be to use modules. However, as these are not first class, we would be unable to define, e.g. homomorphisms between algebraic structures.

The first step of the approach of Spitters and van der Weegen is to define an *operational type class* for each operation and relation.

```
Class Equiv A := equiv: relation A.
Infix "=" := equiv: type_scope.
Notation "(=)" := equiv (only parsing).
Class SgOp A := sg_op: A → A → A.
```

---

[5]An elegant solution is proposed by Pollack [Pol02]. However, its implementation requires simultaneous inductive recursive definitions which are currently not supported in Coq.

Infix "&" := sg_op (at level 50, left associativity).
Notation "(&)" := sg_op (only parsing).

HASKELL-style notations (=) and (&) are defined so operations and relations can easily be used in partially applied position.

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. The class for semigroups is as follows[6].

Class Setoid A {Ae : Equiv A} : Prop := setoid_eq :> Equivalence (@equiv A Ae).
Class SemiGroup A {Ae : Equiv A} {Aop: SgOp A} : Prop := {
  sg_setoid :> @Setoid A Ae;
  sg_op_proper :> Proper ((=) ⟹ (=) ⟹ (=)) (&);
  sg_ass :> Associative (&) }.

Since the operations are unbundled we can easily support sharing. First we make classes for the semiring operations and show that these are in fact special instances of the group operations. For example[7]:

Class Mult A := mult: A → A → A.
Infix "∗" := mult.
Notation "(.∗.)" := mult (only parsing).
Instance mult_is_sg_op '{f : Mult A} : SgOp A := f.

The SemiRing class is then as follows.

Class SemiRing A {Ae : Equiv A} {Aplus : Plus A}
    {Amult : Mult A} {Azero : Zero A} {Aone : One A} : Prop := {
  semiplus_monoid :> @CommutativeMonoid A Ae plus_is_sg_op zero_is_mon_unit;
  semimult_monoid :> @CommutativeMonoid A Ae mult_is_sg_op one_is_mon_unit;
  semiring_distr :> LeftDistribute (.∗.) (+);
  semiring_left_absorb :> LeftAbsorb (.∗.) 0 }.

The syntax :> in the definition of SemiRing declares certain fields as substructures[8], so that in any context where $(A, =, +, *, 0, 1)$ is known to be a SemiRing, $(A, =, +, 0)$ and $(A, =, *, 1)$ are automatically known to be CommutativeMonoids (and so on, transitively, because instance resolution is recursive). In our hierarchy, these substructures by themselves establish the inheritance diagram as in Figure 1.

Without type classes it would be cumbersome to manually carry around the arguments of the class. However, because these arguments are type classes themselves, the type class machinery will perform that job for us. Therefore, all arguments, except the carrier A are declared as implicit using the syntax {x : X}, so the user does not have to specify them.

Proving that an actual structure is an instance of the SemiRing interface is straightforward. First we define instances of the operational type classes.

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

---

[6]We sometimes use the @ prefix to bypass implicit arguments in order to avoid ambiguity.

[7]We use (.∗.) instead of (∗) due to conflicting notations with COQ's comments.

[8]This syntax should not be confused with the similar syntax for coercions in records (e.g. in the bundled representation of a SemiGroup on page 7).

Here we see that instances are just ordinary constants of the class types. However, we use the Instance keyword instead of Definition to let the type class machinery register the instance. Now, to prove that the Peano naturals are in fact a semiring, we just write:

Instance: SemiRing nat.
Proof. ... Qed.

The implicit arguments of SemiRing nat are automatically inferred by instance search. In order to type check SemiRing nat, it has to solve @SemiRing nat ?1 ?2 ?3 ?4 ?5 with obligations ?1 : Equiv nat, ..., ?5 : One nat. Since we have declared instances nat_equiv : Equiv nat, ..., nat_1 : One nat, type class search will trivially solve these obligations. Thus SemiRing nat is actually @SemiRing nat nat_equiv nat_plus nat_mult nat_0 nat_1 with all type class constraints resolved.

The SemiRing type class can be used as follows.

Lemma example '{SemiRing A} x : 1 * x = x + 0.

The backtick instructs COQ to automatically insert implicit declarations, namely Ae Aplus Amult Azero Aone. It also lets us omit a name for the SemiRing A argument itself. All of these arguments will be given automatically generated names that we will never refer to. Furthermore, instance resolution will automatically find instances of the operational type classes for the written notations. Thus the above is really:

Lemma example {A Ae Aplus Amult Azero Aone} {P : @SemiRing A Ae Aplus Amult Azero Aone} (x : A) :
  @equiv A Ae
    (@mult A Amult (@one A Aone) x)
    (@plus A Aplus x (@zero A Azero)).

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 [SvdW11].

Class NaturalsToSemiRing (A : Type) :=
  naturals_to_semiring : ∀ B '{Mult B} '{Plus B} '{One B} '{Zero B}, A → B.
Class Naturals A {Ae Aplus Amult Azero Aone} '{U : NaturalsToSemiRing A} := {
  naturals_ring :> @SemiRing A Ae Aplus Amult Azero Aone;
  naturals_to_semiring_mor :> ∀ '{SemiRing B}, SemiRing_Morphism (naturals_to_semiring A B);
  naturals_initial :> Initial (semirings.object A) }.

These abstract interfaces for the naturals and integers make it easy to change the concrete representation in the future. As fields are not algebraic, no such algebraic specification exists for the rational numbers. Hence, we choose to specify $\mathbb{Q}$ 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 A {Ae : Equiv A} {Azero : Zero A} : Type :=
  frac { num : A; den : A; den_ne_0 : den ≠ 0 }.
Class RationalsToFrac (A : Type) := rationals_to_frac : ∀ B '{Integers B}, A → Frac B.
Class Rationals A {Ae Aplus Amult Azero Aone Aneg Arecip} '{U : !RationalsToFrac A} : Prop := {
  rationals_field :> @DecField A Ae Aplus Amult Azero Aone Aneg Arecip;
  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) }.

In current versions of COQ, inference of substructures is based on *backward* reasoning. In our semiring example that means, each time a CommutativeMonoid A instance is needed,

instance search may try to find a SemiRing A instance. This style of instance search presents some problems, as the following example illustrates.

Class Setoid_Morphism {A B} {Ae : Equiv A} {Be : Equiv B} (f : A → B) := {
   setoidmor_a :> Setoid A;
   setoidmor_b :> Setoid B;
   sm_proper :> Proper ((=) ⟹ (=)) f }.

Each time we have to establish Setoid R for some R, instance search might try to infer a Setoid_Morphism from an arbitrary S to R, or vice versa. Since this search quickly results in a serious blow-up, we omit the substructure declaration :>. Support for *forward* reasoning may solve this problem. If we would be in a context in which we know something to be a Setoid_Morphism, then forward reasoning automatically infers that the source and target are Setoids. Recently, an initial implementation of forward reasoning has been added to Coq, but it suffers from some other performance problems.

4.1. **Constructive fields and apartness.** In constructive mathematics, the common notion of inequality as the negation of equality is often too weak because a proof of a negation lacks computational content. For example, in order to define the reciprocal of $x \in \mathbb{R}$, one needs a witness $\varepsilon \in \mathbb{Q}_{>0}$ that $|x| \geq \varepsilon$. Such a witness cannot be extracted from a proof of $x \neq 0$. To solve this problem, one uses a setoid equipped with an apartness (irreflexive, asymmetric and co-transitive) relation describing inequality [TvD88].

The algebraic hierarchy in the CoRN library [CFGW04] has been built on top of such setoids. Unfortunately, this hierarchy is quite 'heavy' in practice. First, for structures with decidable equality, the negation of equality is the only *tight* apartness. Hence, when working with decidable structures, an apartness relation is unnecessary. Secondly, CoRN uses an *informative* (that is, Type based) apartness relation to facilitate extraction of witnesses. However, Coq's present implementation of setoid rewriting does not support rewriting over relations in Type. So, it does not allow us to replace equations in expressions involving CoRN's informative apartness and thus many proofs involve a lot of manual labor.

To remedy these issues we propose an alternative solution. We use a *non-informative* (that is, Prop-based) apartness relation to enable setoid rewriting and include it just in the parts of the algebraic hierarchy where we actually need it. The latter keeps our interfaces clean and easy to use and should combine the best of two worlds. Type classes are of great help to reduce bookkeeping and clutter in proofs.

Although using a non-informative apartness relation enables setoid rewriting, it disables extraction of witnesses. Fortunately, in case of the reals, a witness can be obtained inefficiently by bounded linear search (see Section 2.3 and 5.1). We think our approach is a reasonable trade-off since the amount of reasoning exceeds the potential use of apartness in computation. In case we need a witness for efficient computation, we just have to specify it explicitly. This approach of specifying witnesses explicitly was already preferred by O'Connor [O'C08], even when an informative apartness was available.

Our interface for a setoid with apartness (henceforth StrongSetoid) is as follows.

Class Apart A := apart: relation A.
Infix "⋈ " := apart (at level 70, no associativity) : type_scope.

Class StrongSetoid A {Ae: Equiv A} {Aap : Apart A} : Prop := {
   strong_setoid_irreflexive :> Irreflexive (⋈) ;
   strong_setoid_symmetric :> Symmetric (⋈) ;

strong_setoid_cotrans :> CoTransitive ($\asymp$) ;
tight_apart : $\forall$ x y, ¬ x $\asymp$ y ↔ x = y }.

This interface is equipped with a *tight* equality. We prove that each StrongSetoid is a Setoid. For decidable structures, we define the following class to describe that the apartness relation is the negation of equality.

Class TrivialApart A '{Equiv A} {ap : Apart A} := trivial_apart : $\forall$ x y, x $\asymp$ y ↔ x $\neq$ y.

Given a setoid with decidable equality we can easily extend it to a StrongSetoid.

Instance default_apart '{Equiv A} : Apart A | 20 := ($\neq$).
Instance default_apart_trivial '{Equiv A} : TrivialApart A (ap:=default_apart).
Lemma dec_strong_setoid '{Setoid A} '{Apart A}
 '{!TrivialApart A} '{$\forall$ x y, Decision (x = y)} : StrongSetoid A.

Unfortunately, the type class mechanism is unable to detect simple loops. Hence we define dec_strong_setoid as an ordinary Lemma instead of an Instance. This trick prevents COQ from using it in instance search and therefore avoids endless derivations of the form StrongSetoid A, Setoid A, StrongSetoid A, ...

For ordinary setoids we want functions to be Proper, which means that they respect equality. For setoids with apartness we need a stronger property, *strong extensionality.*

Class StrongSetoid_Morphism {A B} {Ae : Equiv A} {Aap : Apart A}
 {Be: Equiv B} {Bap : Apart B} (f : A → B):= {
strong_setoidmor_a: StrongSetoid A;
strong_setoidmor_b: StrongSetoid B;
strong_extensionality : $\forall$ x y, f x $\asymp$ f y → x $\asymp$ y }.

We prove that for each StrongSetoid_Morphism f we have Proper ((=)$\Longrightarrow$ (=))f. The only structures for which we actually need apartness are implementations of the real numbers, hence we only base the Field class on top of a StrongSetoid instead of the complete algebraic hierarchy. Our class for fields is as follows. (The PropHolds class is explained in the next subsection.)

Class Recip A '{Apart A} '{Zero A} := recip: { x : A | x $\asymp$ 0 } → A.
Notation "// x" := (recip x).
Notation "(//)" := recip (only parsing).

Class Field A {Ae Aplus Amult Azero Aone Aneg} {Aap : Apart A} {Arecip : Recip A} : Prop := {
field_ring :> @Ring A Ae Aplus Amult Azero Aone Aneg;
field_strongsetoid :> StrongSetoid A;
field_plus_ext :> StrongSetoid_BinaryMorphism (+);
field_mult_ext :> StrongSetoid_BinaryMorphism (.∗.);
field_nontrivial :> PropHolds (1 $\asymp$ 0);
recip_proper :> Setoid_Morphism (//);
recip_inverse : $\forall$ x, proj1_sig x // x = 1 }.

We do not include strong extensionality of the inverse and the reciprocal because these properties can be derived.

For convenience, we define an additional class DecField for fields with decidable equality and whose reciprocal function is total. This class integrates nicely with COQ's rational numbers Q and bigQ, and the field tactic to solve field equations. This total reciprocal function should satisfy $/0 = 0$, so properties as $f(/x) = /(fx)$, $/(/x) = x$ and $/x ∗ /y = /(x ∗ y)$ hold without any additional premises. We proved that a DecField is also an instance of our Field class. A diagram of our complete algebraic hierarchy is displayed in Figure 1.
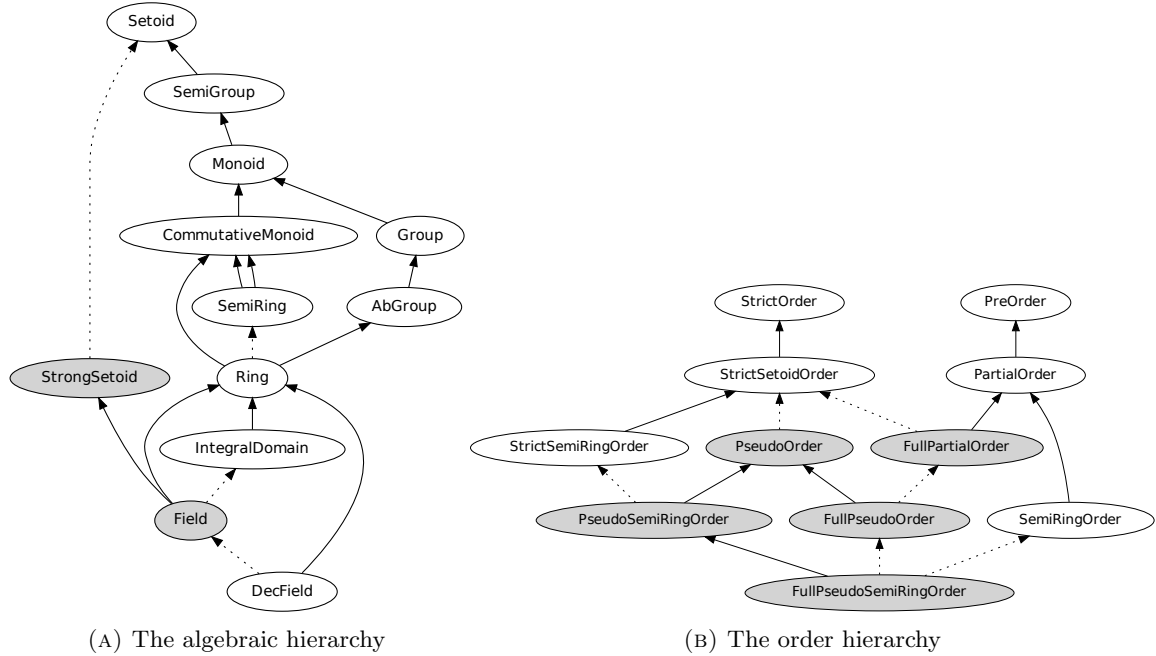
(A) The algebraic hierarchy

(B) The order hierarchy

FIGURE 1. The algebraic and order hierarchy. Dotted lines denote derived inheritance, filled nodes denote presence of apartness.

4.2. **Order theory.** Existing CoQ libraries for ordered algebraic structures turn out to be too limited to abstract from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ and their various implementations. The formalization of ordered fields in the CoRN library [CFGW04] restricts to a very specific part of the algebraic hierarchy (namely fields). Letouzey's NUMBERS library, which is included in recent versions of CoQ trunk, only considers $\mathbb{N}$ and $\mathbb{Z}$. The SSREFLECT library presently restricts to decidable structures with Leibniz equality. Moreover, even mathematically, the most convenient abstraction is not entirely clear. Lešnik [Les10] provides a smooth order theoretic characterization of these structures as so-called *streaks*. We, however, prefer our theory below as it avoids partial functions.

In this work we present a library that captures the notion of order on a variety of structures, including structures with undecidable equality. One of the building blocks of our hierarchy is a pseudo order [Hey56], which is the constructive variant of a total order.

Class PseudoOrder '{Ae : Equiv A} '{Aap : Apart A} (Alt : Lt A) : Prop := {
  pseudo_order_setoid : StrongSetoid A;
  pseudo_order_asym : $\forall$ x y, $\neg$ (x < y $\wedge$ y < x);
  pseudo_order_cotrans :> CoTransitive (<);
  apart_iff_total_lt : $\forall$ x y, x $\asymp$ y $\leftrightarrow$ x < y $\vee$ y < x }.

In case equality is decidable, this interface is rather awkward to work with. Therefore we present ways to go back and forth between the usual classical notions and their constructive variants. For example, we use the type class machinery to infer the classical trichotomy property in case apartness is just the negation of equality.

Instance lt_trichotomy '{PseudoOrder A} '{!TrivialApart A} '{$\forall$ x y, Decision (x = y)} : Trichotomy (<).

Also, we can go the other way around. If we have a StrictSetoidOrder (an ordinary strict order built upon a setoid) satisfying the trichotomy property, we obtain a pseudo order.

Lemma dec_strict_pseudo_order '{StrictSetoidOrder A} '{Apart A}
  '{!TrivialApart A} '{∀ x y, Decision (x = y)} '{!Trichotomy (<)} : PseudoOrder (<).

In order to avoid loops, we define the above as an ordinary Lemma instead of an Instance. Next, one could extend a pseudo order to the standard notion of a (pseudo) ring order.

Class PseudoRingOrder '{Equiv A} '{Apart A} '{Plus A}
    '{Mult A} '{Zero A} '{One A} '{Negate A} (Alt : Lt A) := {
  pseudo_ringorder_spo :> PseudoOrder Alt;
  pseudo_ringorder_ring : Ring A;
  pseudo_ringorder_mult_ext :> StrongSetoid_BinaryMorphism (.∗.);
  pseudo_ringorder_plus :> ∀ z, StrictlyOrderPreserving (z +);
  pseudo_ringorder_mult : ∀ x y, 0 < x → 0 < y → 0 < x ∗ y }.

However, we wish to use our library on ordered structures for implementations of the natural numbers as well. Since the natural numbers do not form a ring, but merely a semiring, we strengthen the above class with a partial subtraction function (living in Prop, because we never use it for computations) and require addition to be order reflecting. We call this, apparently new notion, a PseudoSemiRingOrder.

Class PseudoSemiRingOrder '{Equiv A} '{Apart A} '{Plus A}
    '{Mult A} '{Zero A} '{One A} (Alt : Lt A) := {
  pseudo_srorder_strict :> PseudoOrder Alt;
  pseudo_srorder_semiring : SemiRing A;
  pseudo_srorder_partial_minus : ∀ x y, ¬ y < x → ∃ z, y = x + z;
  pseudo_srorder_plus :> ∀ z, StrictOrderEmbedding (z +);
  pseudo_srorder_mult_ext :> StrongSetoid_BinaryMorphism (.∗.);
  pseudo_srorder_pos_mult_compat : ∀ x y,
    PropHolds (0 < x) → PropHolds (0 < y) → PropHolds (0 < x ∗ y) }.

Instead of including the PseudoRingOrder class in our development, we include a lemma to construct a PseudoSemiRingOrder from a ring satisfying the PseudoRingOrder axioms.

Given a pseudo (semiring) order, one could define the non-strict order $x \leq y$ in terms of the strict order, namely as $\neg y < x$. However, this is quite inconvenient in practice, because we also want to talk about a priori different non-strict orders such as those defined in the standard library. Hence we introduce the following class.

Class FullPseudoSemiRingOrder '{Equiv A} '{Apart A} '{Plus A}
    '{Mult A} '{Zero A} '{One A} (Ale : Le A) (Alt : Lt A) := {
  full_pseudo_srorder_pso :> PseudoSemiRingOrder Alt;
  full_pseudo_srorder_le_iff_not_lt_flip : ∀ x y, x ≤ y ↔ ¬ y < x }.

A diagram of our complete order hierarchy is displayed in Figure 1.

Our theory on abstract orders avoids duplication of theorems and proofs. For example, the following lemmas apply to $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and the dyadics, because all of these structures form a FullPseudoSemiRingOrder.

Lemma plus_compat x_1 y_1 x_2 y_2 : $x_1 \leq y_1$ → $x_2 \leq y_2$ → $x_1 + x_2 \leq y_1 + y_2$.
Lemma lt_1_2 : 1 < 2.
Lemma square_nonneg x : $0 \leq x \ast x$.

To allow us to refer by canonical names to common properties, we introduce classes like:

Class OrderPreserving {A B} {Ae : Equiv A} {Ale : Le A} {Be : Equiv B} {Ble : Le B} (f : A → B) := {

    order_preserving_morphism :> Order_Morphism;
    order_preserving : ∀ x y, x ≤ y → f x ≤ f y }.
Class OrderReflecting {A B} {Ae : Equiv A} {Ale : Le A} {Be : Equiv B} {Ble : Le B} (f : A → B) := {
    order_preserving_back_morphism :> Order_Morphism;
    order_preserving_back : ∀ x y, f x ≤ f y → x ≤ y }.

Here, an Order_Morphism is just the factoring out of the common parts of both classes; namely that f and ≤ respect equality. For the case of multiplication these properties have additional premises, for example:

Global Instance: ∀ (z : A), PropHolds (0 < z) → OrderPreserving (z ∗.).

We introduce the PropHolds class to let the type class machinery prove these properties automatically. For example consider:

Lemma example (n : N) (x y : A) : x ≤ y → (2 ˆ n + 2) ∗ x ≤ (2 ˆ n + 2) ∗ y.
Proof. intros. now apply (order_preserving (2 ˆ n + 2 .∗)). Qed.

In order to use order_preserving, we need a proof of PropHolds (0 < 2 ˆ n + 2). Type class resolution is able to prove this in a fully automated way because we have the following instances:

Instance: PropHolds (0 < 2);
Instance: ∀ x y : A, PropHolds (0 < x) → PropHolds (0 < y) → PropHolds (0 < x + y)
Instance: ∀ (n : N) (x : A), PropHolds (0 < x) → PropHolds (0 < x ˆ n)

This example shows that type class search is in fact very similar to proof search by the auto tactic, but there is no need to call a tactic by hand.

    For arbitrary instances of ℕ, ℤ, ℚ it is easy to define an order satisfying these interfaces:

Instance nat_le '{Naturals N} : Le N | 10 := λ x y, ∃ z, y = x + z.
Instance nat_lt '{Naturals N} : Lt N | 10 := λ x y, x ≤ y ∧ x ≠ y.

However, often we encounter an a priori different order on a structure, most likely an order defined in Coq's standard library (like Nle and Nlt on N). Therefore we prove that a FullPseudoSemiRingOrder uniquely specifies the order on ℕ, ℤ and ℚ. For example:

Context '{Naturals N} '{Naturals N2} {f : N → N2} '{!SemiRing_Morphism f}
    '{Apart N} '{!TrivialApart N} '{!FullPseudoSemiRingOrder (A:=N) Nle Nlt}
    '{Apart N2} '{!TrivialApart N2} '{!FullPseudoSemiRingOrder (A:=N2) N2le N2lt}.
Global Instance: OrderEmbedding f.

Unfortunately Coq has no support to have an argument be 'inferred if possible, generalized otherwise'; see [SvdW11]. When declaring an argument of FullPseudoSemiRingOrder, one is often in a context where most of its components are already available. Here, only the additional arguments Le, Lt and Apart have to be introduced. The current workaround in these cases (as shown above) involves providing names for components that are then never referred to, which is a bit awkward. In the above it would much nicer to write:

Context '{Naturals N} '{Naturals N2} {f : N → N2} '{!SemiRing_Morphism f}
    '{!TrivialApart N} '{!FullPseudoSemiRingOrder N} '{!TrivialApart N2} '{!FullPseudoSemiRingOrder N2}.
Global Instance: OrderEmbedding f.

4.3. **Basic operations.** The operation nat_pow is most commonly, but inefficiently, defined as repeated multiplication and the operation shiftl is defined as repeated duplication. 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, nat_pow and int_pow. 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} '{Le A} '{Zero A} '{Negate A}
   := abs_sig: $\forall$ (x : A), { y : A | (0 $\leq$ x $\rightarrow$ y = x) $\wedge$ (x $\leq$ 0 $\rightarrow$ y = −x)}.
Definition abs '{Abs A} := λ x : A, proj1_sig (abs_sig x).

However, for more complex operations, such as shiftl, we follow the unbundled approach by Spitters and van der Weegen [SvdW11].

Class ShiftL A B := shiftl: A $\rightarrow$ B $\rightarrow$ A.
Infix "$\ll$" := shiftl (at level 33, left associativity).
Class ShiftLSpec A B (sl : ShiftL A B) '{Equiv A} '{Equiv B} '{One A}
   '{Plus A} '{Mult A} '{Zero B} '{One B} '{Plus B} := {
  shiftl_proper : Proper ((=) $\implies$ (=) $\implies$ (=)) ($\ll$) ;
  shiftl_0 :> RightIdentity ($\ll$) 0;
  shiftl_S : $\forall$ x n, x $\ll$ (1 + n) = 2 ∗ x $\ll$ 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. Since theory on shifting with exponents in $\mathbb{N}$ and $\mathbb{Z}$ is similar we want to avoid duplication of theorems and proofs. To this end we introduce a class describing the bi-induction principle.

Class Biinduction A '{Equiv A} '{Zero A} '{One A} '{Plus A} : Prop
  := biinduction (P: A $\rightarrow$ Prop) '{!Proper ((=) $\implies$ iff) P} : P 0 $\rightarrow$ ($\forall$ n, P n $\leftrightarrow$ P (1 + n)) $\rightarrow$ $\forall$ n, P n.

Since this class is inhabited by any integer and natural implementation we can parametrize theory on shiftl as follows.

Context '{SemiRing A} '{!LeftCancellation (.∗.) (2:A)} '{SemiRing B} '{!Biinduction B} '{!ShiftLSpec A B sl}.
Lemma shiftl_base_plus x y n : (x + y) $\ll$ n = x $\ll$ n + y $\ll$ n.
Global Instance shiftl_inj: $\forall$ n, Injective ($\ll$n).

4.4. Decision procedures.
The Decision type class by Spitters and van der Weegen collects decidable propositions [SvdW11].

Class Decision P := decide: sumbool P ($\neg$ P).

Using this type class we can declare a argument '{$\forall$ 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 us to easily compose deciders, for example:

Instance prod_dec '(A_dec : $\forall$ x y : A, Decision (x = y))
  '(B_dec : $\forall$ x y : B, Decision (x = y)) : $\forall$ x y : A ∗ B, Decision (x = y).

We have to be careful however. Consider the definition of the order on the dyadics.

Global Instance dy_le: Le Dyadic := λ x y : Dyadic,
  ZtoQ (mant x) ∗ 2 ˆ (expo x) $\leq$ ZtoQ (mant y) ∗ 2 ˆ (expo y)
Global Instance dy_le_dec: $\forall$ (x y : Dyadic), Decision (x $\leq$ y).

Now, decide (x $\leq$ y) for x and y of type Dyadic is actually @decide (x $\leq$ y) (dy_le_dec x y). This shows that the proposition x $\leq$ y is just a phantom argument used for instance search only, whereas dy_le_dec is the decision procedure doing the actual work. Due to eager evaluation of CoQ's virtual machine, the term decide (x $\leq$ y) is expanded to

@decide (ZtoQ (mant x)∗ 2 ˆ (expo x)≤ ZtoQ (mant y)∗ 2 ˆ (expo y)) (dy_le_dec x y),

resulting in the phantom argument being evaluated first. In many cases evaluation of such a phantom argument is cheap, but here it involves an expensive conversion of x and y to Q. We avoid evaluation of this phantom argument by wrapping it under 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.

Now, if we write decide_rel ($\leq$) x y, it expands to

@decide_rel (λ x y, ZtoQ (mant x)∗ 2 ˆ (expo x)≤ ZtoQ (mant y)∗ 2 ˆ (expo y)) x y dy_le_dec,

where the definition of inequality is safely hidden under a λ-abstraction.

This problem would not appear if COQ's virtual machine would evaluate propositions lazily, as the phantom argument is just a proposition. Unfortunately, lazy evaluation of propositions is not supported by its current implementation.

4.5. **Explicit type casts.** The Cast type class collects (explicit) type casts.

Class Cast A B := cast: A → B.
Implicit Arguments cast [[Cast]].
Notation "' x" := (cast _ _ x) (at level 20).
Instance: Params (@cast) 3.

This definition allows us to refer to a cast from x : A to B by using an apostrophe, or writing cast A B x. An example of an instance of this class is:

Instance NonNeg_inject: Cast ($A_{\geq 0}$) A := @proj1_sig A _.

Here, $A_{\geq 0}$ is a Σ-type describing the non-negative cone of an ordered ring A. Contrary to COQ's built-in coercion mechanism, our type casts are explicit instead of implicit and type classes are used to register them. Our approach has some advantages:

(1) By using type classes to register casts, we are allowed to parametrize classes with casts. An example is the AppRationals class, as defined in Section 5.
(2) Implicit coercions often introduce ambiguity. Since our approach allows us to refer to casts by a (canonical) name, e.g. cast B C (cast A B x), we can avoid this ambiguity.
(3) Casts can be put in partially applied position, e.g. order_preserving (cast Z Q).

COQ's coercion mechanism does not allow us to define a coercion from $A_{\geq 0}$ to A nor a coercion from a ring to its polynomial ring. More generally, it does not allow most forms of parametrized coercions nor non-uniform coercions. An implementation that allows parametrized coercions like NonNeg_inject has to avoid an infinite loop: to naively type check x : A, one has to type check x : $A_{\geq 0}$, x : $(A_{\geq 0})_{\geq 0}$, . . . We suffer from such loops if we compose our Cast classes automatically as well. Hence we refrain from adding:

Instance cast_comp_base '{f : Cast A B} : ComposedCast A B := f.
Instance cast_comp_step '{f : Cast B C} '{g : ComposedCast A B} : ComposedCast A C := λ x, f (g x).

Matita [ASCTZ07] allows parametrized coercions and avoids the loop by not applying coercions recursively, but instead building a well-chosen set of set of composite coercions [Tas08]. Non-uniform coercions [ST11] are available in Matita. They are implemented using unification hints, a feature similar to type classes.

5. The real numbers

To make our implementation of the reals independent of the underlying dense set, we provide an abstract specification of *approximate rationals* inspired by the notion of *approximate fields* — a field with approximate operations — which is used in Bauer and Kavler's RZ implementation of the exact reals [BK08]; see also [BT09]. In particular, we provide an implementation of this interface by dyadics based on Coq's machine integers.

Our interface for approximate rationals 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. For efficient computation we include the operations: approximate division, normalization, an embedding of Z, absolute value, power by N, shift by Z, and decision procedures for equality and order.

Class AppDiv AQ := app_div: AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx: AQ → Z → AQ.
Class AppRationals AQ {AQe AQplus AQmult AQzero AQone AQneg} '{Apart AQ} '{Le AQ} '{Lt AQ}
    {AQtoQ : Cast AQ Q_as_MetricSpace} '{!AppInverse AQtoQ} {ZtoAQ : Cast 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 AQe AQplus AQmult AQzero AQone AQneg;
  aq_trivial_apart :> TrivialApart AQ;
  aq_order_embed :> OrderEmbedding AQtoQ;
  aq_strict_order_embed :> StrictOrderEmbedding AQtoQ;
  aq_ring_morphism :> SemiRing_Morphism AQtoQ;
  aq_dense_embedding :> DenseEmbedding AQtoQ;
  aq_div : ∀ x y k, ball (2 ˆ k) ('app_div x y k) ('x / 'y);
  aq_compress : ∀ x k, ball (2 ˆ k) ('app_approx x k) ('x);
  aq_shift :> ShiftLSpec AQ Z (≪) ;
  aq_nat_pow :> NatPowSpec AQ N (ˆ);
  aq_ints_mor :> SemiRing_Morphism ZtoAQ }.

We define the real numbers as the completion of the approximate rationals. To create functions on the real numbers, we use the monadic operations bind or map. This approach is convenient because equality and inequality are decidable on the approximate rationals, whereas it is not on the real numbers. For binary functions, e.g. addition and multiplication, we use the map2 function, as described in [O'C07].

O'Connor [O'C07] keeps the size of the rational numbers small to avoid efficiency problems. He introduces a function approx x ϵ that yields the 'simplest' rational number between $x - \epsilon$ and $x + \epsilon$. 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 x ϵ, app_approx x (Qdlog2 ϵ)) such that compress x = x.

In Section 5.4 we will explain our choice of using a power of 2 to specify the precision of app_div and app_approx.

5.1. **Order and apartness.** Following [BB85, O'C09], we define non-negativity and the order on the real numbers as follows.

$$\text{NonNeg } x := \forall \varepsilon : \mathbb{Q}_{>0}, -\varepsilon \leq x \; \varepsilon$$
$$x \leq y := \text{NonNeg } (y - x)$$

Bishop and Bridges [BB85] define positivity as the dual of non-negativity: $\exists \varepsilon : \mathbb{Q}_{>0}, \ \varepsilon < x \ \varepsilon$. O'Connor [O'C09] defines positivity and the strict order differently so as to avoid a potentially expensive computation, namely $x \ \varepsilon - \varepsilon$, to obtain a witness between 0 and x.

$$\mathsf{Pos} \ x := \{\varepsilon : \mathbb{Q}_{>0} \mid \varepsilon \leq x\}$$

$$x <_{\mathbf{T}} y := \mathsf{Pos} \ (y - x)$$

We use the $\mathbf{T}$ subscript to emphasize that the relation lives in Type. Next, we define $x \asymp_{\mathbf{T}} y := x <_{\mathbf{T}} y \vee y <_{\mathbf{T}} x$. Extraction of a witness $\varepsilon \in (0, x]$ from $\mathsf{Pos} \ x$ allows us to define the reciprocal function of type $\forall x : \mathbb{R}, 0 \asymp_{\mathbf{T}} x \to \mathbb{R}$.

In order to use our type class based hierarchy we need a strict order and apartness relation in Prop. We need this restriction because Coq's present implementation of setoid rewriting does not allow rewriting in Type-based relations (see Section 4.1). Our definition is similar to Bishop and Bridges' definition of positivity, but uses shifts instead.

$$x < y := \exists n : \mathbb{N}, \ 1 \ll -n < (y - x) \ (1 \ll (-n - 1))$$

$$x \asymp y := x < y \vee y < x$$

Using constructive indefinite description (see Section 2.3), it is an easy job to prove that we indeed have $x < y \leftrightarrow x <_{\mathbf{T}} y$ and $x \asymp y \leftrightarrow x \asymp_{\mathbf{T}} y$. Similar to O'Connor [O'C09], we implement a tactic that automatically proves strict inequalities. The tactic terminates iff the inequality holds and is similar to our use of linear search to obtain $x <_{\mathbf{T}} y$ from $x < y$.

5.2. **Implementation using 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 a specific implementation of integers, we have defined most of the operations for arbitrary integer implementations. Given such an implementation Int it is straightforward to define 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: Cast Int Dyadic := λ x, x ▼ 0.
Global Instance dy_negate: Negate Dyadic := λ x, −mant x ▼ expo x.
Global Instance dy_mult: Mult Dyadic := λ x y, mant x ∗ mant y ▼ expo x + expo y.
Global Instance dy_0: Zero Dyadic := cast Int Dyadic 0.
Global Instance dy_1: One Dyadic := cast Int Dyadic 1.
Global Program Instance dy_plus: Plus 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 $(\ll)$ has type $\mathsf{Int} \to \mathsf{Int}_{\geq 0} \to \mathsf{Int}$, where $\mathsf{Int}_{\geq 0}$ is a Σ-type describing the non-negative cone 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.

The operation of approximate division is not implemented in an abstract way as we have not developed a type class and theory for right shifts yet. For our implementation using Coq's machine integers bigZ, we defined approximate division concretely using the shift right function from the standard library.

5.3. **Implementation using the rationals.** Our development contains additional implementations of the AppRationals class using COQ's old rational numbers Q and the new rational numbers bigQ (which are built from the machine integers bigZ). Although creating these implementations is uninteresting from a performance point of view, it confirms that it is trivial to change the underlying dense set from which our real numbers are built.

To implement the app_approx function in an efficient manner, we use shifts on the underlying integers. Furthermore, to keep the size of the results of the division operation small, we incorporate the app_approx function.

Instance bigQ_div: AppDiv bigQ := λ x y, app_approx (x / y).

5.4. **Power series.** Elementary transcendental functions as exp, sin, ln and atan can be defined by their power series. If the coefficients of a power series are alternating, decreasing and have limit 0, then we obtain a fast converging sequence with an easy termination proof. For $-1 \leq x \leq 0$,

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

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 $1, a, a^2, a^3, \ldots$ 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 (and guarded co-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 O'Connor [O'C09] uses LazyExists.

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.

Unfortunately, our experiments showed that the above still yields too much overhead due unnecessary to reduction of proofs. To remedy this issue we introduce the following function where Str_nth_tl n s takes the n-th tail of the stream s.

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.
```

This function adds n additional LazyFurther constructors. When instantiated with a big enough n, computation will suffer from the implementation limits of CoQ (e.g. a stack overflow) or runs out of memory, before it ever refers to the actual proof. Using LazyExists_inc we are able to compute on average twice the amount of decimals as we did before on examples such as the ones in Table 2.

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 extend this in two ways. First, we generalize various notions to abstract structures. Secondly, as we do not have exact division on approximate rationals, we extend the algorithm to work with approximate division. The latter requires 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 a k such that:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\left(\text{app\_div } n_k\ d_k\ \left(\log\frac{\varepsilon}{2k}\right) + \frac{\varepsilon}{2k}\right)\ 0. \tag{5.1}$$

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 5, we have specified the precision of division in powers of 2 instead of using a rational value. This allows us to replace (5.1) with:

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

Here k is the length of the partial sum, and 2^l, where $l = \log \varepsilon - (k+1)$, 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 a few extra terms have to be computed and due to the approximate division the auxiliary results are kept as small as possible.

Similarly, using this method to compute infinite alternating sums, we use the following series to implement atan x and sin x for $x \in [-1, 1]$.

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

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

We extend these functions to their complete domain by repeatedly applying the following formulas [O'C09].

$$\mathsf{exp}\ x = \left(\mathsf{exp}\ (x \ll 1)\right)^2 \tag{5.2}$$

$$\mathsf{exp}\ x = \frac{1}{\mathsf{exp}\ (-x)} \tag{5.3}$$

$$\mathsf{sin}\ x = 3 * \mathsf{sin}\ \frac{x}{3} - 4 * \left(\mathsf{sin}\ \frac{x}{3}\right)^3 \tag{5.4}$$

$$\mathsf{atan}\ x = -\mathsf{atan}\ (-x) \tag{5.5}$$

$$\mathsf{atan}\ x = \frac{\pi}{2} - \mathsf{atan}\ \frac{1}{x} \quad \text{for } 0 < x \tag{5.6}$$

$$\mathsf{atan}\ x = \frac{\pi}{4} - \mathsf{atan}\ \left(\frac{x-1}{x+1}\right) \quad \text{for } 0 < x \tag{5.7}$$

Since we do not have exact division on the approximate rationals, we parameterize infinite sums by two streams in Equation 5.4, 5.6 and 5.7.

The series described in this section converge faster for arguments closer to 0. We use Equation 5.2 and 5.4 repeatedly to reduce the input to a value $|x| \in [0, 2^k)$. For $50 \leq k$, this yields nearly always major performance improvements, for higher precisions setting it to $75 \leq k$ yields even better results. Unfortunately, we are unaware of a similar trick for atan. We define π in terms of atan using the following Machin-like formula.

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

Again, here we notice the purpose of parameterizing infinite sums by two streams. We define cos in terms of sin.

$$\mathsf{cos}\ x = 1 - 2 * \left(\mathsf{sin}\ \frac{x}{2}\right)^2$$

O'Connor [O'C07, O'C09] subtracts multiples of 2π to reduce the arguments of sin and cos. In our tests this did not lead to performance improvements because our implementation of π turned out to be slower than the performed range reductions.

5.5. **Square root.** We use Wolfram's algorithm [Wol02, p.913] for computing the square root. Its complexity is linear, in fact it provides a new binary digit in each step.

```
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.
```

We write (r_n, s_n) for the n-th pair of approximations. By induction we obtain:

$$s_n^2 + 4r_n = 4^{n+1}a \tag{5.8}$$

$$r_n \le 2s_n + 4 \tag{5.9}$$

$$2^m s_n \le s_{n+m} \le 2^m(s_n + 4) - 4 \tag{5.10}$$

$$r_n \le 2^{3+n} \tag{5.11}$$

By 5.8, $(2^{-(n+1)}s_n)^2 + 2^{-2n}r_n = a$. By 5.11, $2^{-2n}r_n$ converges to 0 as n tends to ∞. Therefore, by 5.10, $2^{-(n+1)}s_n$ is a Cauchy sequence which moreover converges to $\sqrt{a}$.

We extend the square root to its entire domain by repeatedly applying:

$$\sqrt{x} = 2 * \sqrt{\frac{x}{4}}$$

O'Connor's Coq implementation [O'C08] includes the much faster Newton iteration, whose complexity is logarithmic in the number of decimals. The function to iterate is:

Definition f (r : Q) : Q := r / 2 + a / (2 * r).

Because of the absence of exact division on our approximate rationals we cannot implement this function directly. However, we can implement it on our real numbers. As the above definition does not use sharing, we have to define this function on the reals by first defining:

Definition f (r : AQ) (ϵ : Qpos) : AQ := (r + approx_div (Qdlog2 ϵ) a r) $\ll$ (−1).

and then showing that it gives rise to a continuous function $f : \mathsf{AQ} \to \mathsf{AR}$ which we finally lift to a function bind $f : \mathsf{AR} \to \mathsf{AR}$ on the reals. In this way we take care of sharing, division and intermediate use of the approx function (see Section 5) all in one go. We hope the future correctness proof to be quite smooth, since we work with *exact* real numbers. We have implemented this in Haskell and it performs really well.

6. 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 [O'C07]. The second step was to implement and verify this prototype in Coq. Our Coq development contains verified versions of: the field operations, exponentiation by a natural, computation of power series, exp, atan, sin, cos, π and the square root.

In this section we present some benchmarks, taken from the 'Many Digits' friendly competition [NW09], 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 Debian GNU/Linux. The sources of our developments can be found at https://github.com/c-corn/corn.

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 generally more than a 100 times for small examples already. This difference between the comparison of the Haskell and the Coq implementation is explained by the fact that O'Connor's Haskell implementation already uses rational numbers

	Expression	Decimals	Old	New
P01	$\sin\,(\sin\,(\sin\,1))$	5.000	25s	2.3s
P02	$\sqrt{\pi}$	5.000	3.3s	1.7s
P03	$\sin\,e$	5.000	13s	1.2s
P04	$\exp\,(\pi * \sqrt{163})$	5.000	22s	2.0s
P05	$\exp\,(\exp\,e)$	5.000	43s	2.6s
P06	$\log\,(1 + \log\,(1 + \log\,(1 + \log\,(1 + \pi))))$	500	107s	2.5s
P07	$\exp\,1000$	20.000	1.1s	0.7s
P08	$\cos\,(10^{50})$	20.000	6.7s	1.4s
P09	$\sin\,(3 * \frac{\log\,640320}{\sqrt{163}})$	5.000	33s	16s
P11	$\tan\,e + \mathsf{atan}\,e + \tanh\,e + \mathsf{atanh}\,\frac{1}{e}$	500	41s	3.2s
P12	$\mathsf{asin}\,\frac{1}{e} + \cosh\,e + \mathsf{asinh}\,e$	500	99s	3.2s

TABLE 1. HASKELL, compiled with `ghc` version 6.12.1, using `-O2`. The column 'old' refers to the HASKELL prototype of O'Connor, and the column 'new' to our HASKELL prototype.

	Expression	Decimals	Old	New	Decimals	New
P01	$\sin\,(\sin\,(\sin\,1))$	25	46s	0.6s	500	3.8s
P02	$\sqrt{\pi}$	25	0.3s	0.03s	500	6.8s
P03	$\sin\,e$	25	36s	0.1s	500	1.9s
P04	$\exp\,(\pi * \sqrt{163})$	10	214s	0.1s	500	3.7s
P05	$\exp\,(\exp\,e)$	10	36s	0.2s	500	3.2s
P07	$\exp\,1000$	10	2662s	1.0s	2.000	4.9s
P08	$\cos\,(10^{50})$	25	11s	0.3s	2.000	12s

TABLE 2. COQ trunk, revision 14023. The column 'old' refers to the COQ implementation of O'Connor, and the column 'new' to our COQ implementation. Computations using a higher precision did not terminate within a reasonable time using O'Connor's implementation, so these are omitted.

built from fast integers and incorporates various optimizations, while his COQ implementation does not. The last column of Table 2 indicates that our new implementation is able to compute an order of magnitude more decimals in the same amount of time.

We also compared the new reals built from COQ's fast rationals (Section 5.3) and our dyadic rationals (Section 5.2). For $\exp$, $\sin$ and $\cos$ we obtain quite similar results due to the our range reductions to reduce the length of the power series. In case of the square root, the dyadics rationals are much faster because wolfram iteration is designed for an efficient shift. It is interesting to notice that π and atan benefit the least from our improvements, as we are unaware of range reductions to reduce the length of the series.

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 7.4s using type classes and 7.2s without type classes. Here we exclude the time spend on type class resolution. Thus type classes cause only a 3% performance penalty on computations, which is very acceptable for the modularity that they introduce.

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 5.4s.

7. 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).

Canonical structures provide an alternative, and partially complementary, implementation of type classes [GZND11]. By choice, canonical structures restrict to deterministic proof search, this makes them more efficient, but also somewhat more intricate to use. The use of canonical structures by the SSREFLECT team [GGMR09] makes it plausible that with some effort we could have used canonical structures for our work instead. However, the SSREFLECT-library is currently not suited for setoids which are crucial to us. The integration of unification hints [ARCT09] into COQ may allow a tighter integration of type classes and canonical structures.

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. Moreover, setoid rewriting over relations in Type would have made our work much easier.

The experimental native_compute by Boespflug, Dénès and Grégoire [BDG11] 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 an additional speed up of 3 times compared to vm_compute.

The FLOCQ project [BM11] formalizes infinitary floating-points in COQ. It provides a library of theorems on 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 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 [JP09], using the framework described in [Ber07, Jul08], 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 a similar time. On the other hand, the efficiency of π in their framework is comparable with ours. Berger [Ber09], too, uses co-induction for exact real computation.

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. To do so we need to extend the current technology to functional analysis. For instance we will build a type class interface for metric spaces in order to treat various function spaces.

Cohen and Mahboubi [Coh12, CM12] provide a formalization of quantifier elimination for the theory of decidable real closed fields, and an implementation of the algebraic real numbers. Quantifier elimination will automate many proofs in constructive analysis which only involve algebraic real numbers. Conversely, our implementation could be used for efficiently evaluating a Cauchy representation of an algebraic real number.

Acknowledgements. We thank Eelis van der Weegen for many discussions and Pierre Letouzey and Matthieu Sozeau for closing some of our bug reports. We are grateful to the anonymous referees who helped to improve the presentation of the paper.

References

[AGST10]   M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In *ITP*, volume 6172 of *LNCS*, pages 83–98, 2010.

[ARCT09]   A. Asperti, W. Ricciotti, C. Coen, and E. Tassi. Hints in Unification. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 84–98, 2009.

[ASCTZ07] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[BB85]   E. Bishop and D. Bridges. *Constructive analysis*, volume 279 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag, 1985.

[BC04]   Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in TCS. Springer, 2004.

[BDG11]   M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In *CPP*, volume 7086 of *LNCS*, pages 362–377, 2011.

[Ber07]   Y. Bertot. Affine functions and series with co-inductive real numbers. *MSCS*, 17(1):37–63, 2007.

[Ber09]   U. Berger. From coinductive proofs to exact real arithmetic. In *CSL*, volume 5771 of *LNCS*, pages 132–146, 2009.

[Bis67]   E. Bishop. *Foundations of constructive analysis*. McGraw-Hill, 1967.

[BK08]   A. Bauer and I. Kavkler. Implementing real numbers with RZ. *ENTCS*, 202:365–384, 2008.

[BM11]   S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic*, 2011.

[BT09]   A. Bauer and P. Taylor. The dedekind reals in abstract stone duality. *MSCS*, 19(4):757, 2009.

[CFGW04]   L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In *MKM*, volume 3119 of *LNCS*, pages 88–103, 2004.

[CFL06]   L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. *ENTCS*, 151(1):75–91, 2006.

[CFS03]   L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In *TPHOLs*, volume 2758 of *LNCS*, pages 205–220, 2003.

[CH88]   T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[CM12]   C. Cohen and A. Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, 2012.

[Coh12]   C. Cohen. Construction of real algebraic numbers in Coq. In *ITP*, volume 7406 of *LNCS*, pages 67–82, 2012.

[Coq12]   Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, 2012.

[CP90]   T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88*, volume 417 of *LNCS*, pages 50–66, 1990.

[CS12]   P. Castéran and M. Sozeau. A gentle introduction to type classes and relations in Coq, 2012.

[GGMR09]   F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 327–342, 2009.

[GL02]   B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.

[GPWZ02]   H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in coq. *Journal of Symbolic Computation*, 34(4):271–286, 2002.

[GZND11]   G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175, 2011.

[Hal02]   T. C. Hales. A computer verification of the Kepler conjecture. In *The International Congress of Mathematicians, Vol. III*, pages 795–804, 2002.

[Hey56]   Heyting, A. *Intuitionism. An introduction.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1956.

[Hof97]   M. Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS Distinguished Dissertations. Springer, 1997.

[JP09]   N. Julien and I. Pasca. Formal Verification of Exact Computations Using Newton's Method. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 408–423, 2009.

[Jul08]   N. Julien. Certified Exact Real Arithmetic Using Co-induction in Arbitrary Integer Base. In *FLOPS*, volume 4989 of *LNCS*, pages 48–63, 2008.

[Les10]   D. Lesnik. *Synthetic Topology and Constructive Metric Spaces.* PhD thesis, University of Ljubljana, 2010.

[Let08]   P. Letouzey. Extraction in Coq: An Overview. In *CiE*, volume 5028 of *LNCS*, pages 359–369, 2008.

[ML82]   P. Martin-Löf. 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*, pages 153–175, 1982.

[ML98]   P. Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. OUP, 1998.

[Mog89]   E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, 1989.

[NW09]   M. Niqui and F. Wiedijk. 'Many Digits' Friendly Competition. Technical Report ICIS-R09007, Radboud University Nijmegen, 2009.

[O'C07]   R. O'Connor. A Monadic, Functional Implementation of Real Numbers. *MSCS*, 17(1):129–159, 2007.

[O'C08]   R. O'Connor. Certified Exact Transcendental Real Number Computation in Coq. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 246–261, 2008.

[O'C09]   R. O'Connor. *Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory.* PhD thesis, Radboud University Nijmegen, 2009.

[OS10]   R. O'Connor and B. Spitters. A computer verified, monadic, functional implementation of the integral. *TCS*, 411(37):3386–3402, 2010.

[Pal09]   E. Palmgren. Constructivist and Structuralist Foundations: Bishop's and Lawvere's Theories of Sets. Technical Report 4, Mittag-Leffler, 2009.

[Pol02]   R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386––402, 2002.

[Ric08]   F. Richman. Real numbers and other completions. *Mathematical Logic Quarterly*, 54(1):98–108, 2008.

[SO08]   M. Sozeau and N. Oury. First-class type classes. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 278–293, 2008.

[Soz09]   M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.

[Spi11]   A. Spiwack. *Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory.* PhD thesis, INRIA, 2011.

[ST11]   C. Sacerdoti Coen and E. Tassi. Nonuniform coercions via unification hints. In *TYPES*, volume 53 of *EPTCS*, pages 16–29, 2011.

[SvdW11]   B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.

[Tas08]   E. Tassi. *Interactive Theorem Provers: issues faced as a user and tackled as a developer.* PhD thesis, University of Bologna, 2008.

[Tuc02]   W. Tucker. A rigorous ODE solver and smale's 14th problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.

[TvD88]   A. Troelstra and D. van Dalen. *Constructivism in Mathematics: an Introduction.* Number 121 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.
[Wad92]   P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.
[Wol02]   S. Wolfram. *A new kind of science.* Wolfram Media, 2002.