Implementing real numbers with RZ

Andrej Bauer Iztok Kavkler

Faculty of Mathematics and Physics
University of Ljubljana
Slovenia

April 12, 2007

Abstract

RZ is a tool which translates axiomatizations of mathematical structures to program specifications using the realizability interpretation of logic. This helps programmers correctly implement data structures for computable mathematics. RZ does not prescribe a particular method of implementation, but allows programmers to write efficient code by hand, or to extract trusted code from formal proofs, if they so desire. We used this methodology to axiomatize real numbers and implemented the specification computed by RZ. The axiomatization is the standard domain-theoretic construction of reals as the maximal elements of the interval domain, while the implementation closely follows current state-of-the-art implementations of exact real arithmetic. Our results shows not only that the theory and practice of computable mathematics can coexist, but also that they work together harmoniously.

1 Introduction

At Computability and Complexity in Analysis 2003 [7], the first author was asked to suggest open problems and research directions in constructive and computable mathematics. One of the suggestions was:

> *"Suggestion 2: Get closer to practice*
>
> *Put more emphasis on* actual *programming.*
> *(Turing machines are fine, but you can't buy one.)*
>
> *But do not cheat! The relationship between constructive mathematics and programming should be mathematically rigorous."*

The suggestion asks *not* that everyone should forget constructive and computable mathematics and start programming their own exact real arithmetic, but rather implies that it would be useful to strengthen the connection between theory and practice. In order to validate such a claim, we would have to

1. move theory closer to practice by using models of computability which refer to actual programming languages instead of (Type 2) Turing machines, and

2. move practice closer to theory by making sure that practical implementations follow *formal* specifications that are computed directly from theoretical models.

For the first item we have in mind something like a theory of representations in which Type 2 Turing machines are replaced by a real-world programming language. This should not affect the fundamental results of computable mathematics, since general-purpose programming languages are as powerful as Turing machines (and they also support infinite streams and non-terminating computations). For the second item, we would like to see implementations of exact real arithmetic supported by *formal* specifications that at least in principle allow (automatic or human-assisted) checking of correctness. Such specifications should be generated from (formal) descriptions of theoretical models on which the implementations are based.

Taking our own suggestion seriously, we implemented exact real arithmetic while making sure that the connection between the constructive theory of reals and the practical implementation of exact real arithmetic is explicit and mathematically rigorous. In this paper we report on the experience.

The results of a project along these lines can be judged as successful only if theory and practice turn out to help each other, rather than impose unnatural constraints on each other. For example, it would be unacceptable if a programmer adhering closely to a mathematical model were forced to write inefficient or useless code. Conversely, we would not want to sacrifice mathematical elegance just to accommodate programming tricks. Much to our satisfaction, our project shows that constructive and computable theories of reals are indeed in harmony with state-of-the-art implementations of real arithmetic. We are hopeful that in the future our methodology will help develop implementations of other, more advanced computable structures.

The paper is organized as follows. In Section 2 we describe the tools that we used. In Section 3 we discuss our implementation of exact real arithmetic. In Section 4 we evaluate our achievements and suggest directions for future work.

2 Realizability and RZ

Algorithms and data structures in computable mathematics are usually expressed in terms of Turing machines and Gödel encodings (either by numbers or by infinite sequences). This is a natural choice when one considers theoretical questions about computability, but in an actual implementation we use structured programming with datatypes, classes, and other programming constructs. In order to keep mathematics and programming close to each other, we replaced the customary Type 2 representations with representations in a programming language, in our case Objective Caml [11], but other languages could be used. Then we used *RZ* [5], a tool written in a related project by Chris Stone and Andrej Bauer, to automatically translate the constructive theory of reals to a formal program specification. Finally, we implemented the specification in Objective Caml.

It is beyond the scope of this paper to fully describe how RZ works. We refer to [4] for resources on RZ, and to [3, 2] for background on how realizability theory is used to connect constructive and computable mathematics. In this section we explain enough to make the rest of the paper comprehensible.

A *representation* in a programming language[1] $\mathcal{P}$ is a triple (A, τ_A, δ_A) where A is the represented set, τ_A is a type in $\mathcal{P}$ of representing values, and $\delta_A : [\![\tau_A]\!] \rightharpoonup A$ a partial surjection. Here $[\![\tau_A]\!]$ denotes the set of values of type τ_A.[2] The usual Type 2 representations [20] are a special case of representations in which $[\![\tau_A]\!]$ is fixed as the set $\Sigma^{\mathbb{N}}$ of infinite sequences over an alphabet Σ. This difference is not as essential as it may seem, because Type 2 representations are typically described in such a way that the intended type τ_A can be recognized as a suitable subset of $\Sigma^{\mathbb{N}}$.

A representation (A, τ_A, δ_A) determines a *partial equivalence relation (per)* $\approx_A$ on $[\![\tau_A]\!]$, given by

$$u \approx_A v \iff \exists\, x \in A \,.\, \delta_A(u) = \delta_A(v) = x \;.$$

The relation $\approx_A$ need not be reflexive because δ_A need not be defined everywhere. It is convenient to define the *support* $\|A\|$ as the set of those values that represent something,

$$\|A\| = \{u \in [\![\tau_A]\!] \mid u \approx_A u\}.$$

The representation δ_A may be recovered from $\approx_A$ up to isomorphism if we take the represented set to be the equivalence classes of $\approx_A$ and δ_A the canonical quotient map. Thus a representation (A, τ_A, δ_A) may be viewed equivalently as a per $(\tau_A, \approx_A)$. RZ uses pers because they refer only to types and values of $\mathcal{P}$, rather than to arbitrary represented sets.

Pers form a category in which a morphism $f : (\tau_A, \approx_A) \to (\tau_B, \approx_B)$ is represented by a value $f \in [\![\tau_A \to \tau_B]\!]$ which is *extensional* with respect to $\approx_A$ and $\approx_B$, meaning that $u \approx_A v$ implies $f(u) \approx_B f(v)$ for all $u, v \in [\![\tau_A]\!]$. Two such extensional values f_1 and f_2 represent the same morphism when $u \approx_A v$ implies $f_1(u) \approx_B f_2(v)$ for all $u, v \in [\![\tau_A]\!]$. RZ uses the realizability interpretation of constructive logic and dependent type theory in the category of pers to compute specifications from mathematical theories, as is explained in [5].

RZ takes as input one or more *theories* which are written in the usual first-order logic with a rich assortment of set constructions (dependent products and sums, function spaces, subsets, quotients, but no powersets). A theory comprises a list of *declarations* and *definitions* of sets, constants, predicates and relations, as well as *axioms*. For example, Figure 1 shows the theory of a commutative group. To save the world from yet another syntax, RZ mostly follows the syntax of the proof assistant Coq [6] (the relationship between RZ and Coq is discussed in Section 4). For better readability we here display symbols such as `forall`, `exists`, `\/`, `/\`, `->` as $\forall$, $\exists$, $\vee$, $\wedge$, $\to$, respectively.

The first four lines of `CommutativeGroup` are declarations (the `Parameter` keyword) of a set `s`, a constant `zero`, a binary operation `add`, and a unary operation `neg` on `s`. The `Implicit Type` declaration says that, unless otherwise specified, variables `x`, `y` and `z` are presumed to range over `s`. With the `Definition` keyword we define a binary operation `sub`, while the four axioms say that `s`, `zero`, `add` and `neg` together form a commutative group.

[1]In our case $\mathcal{P}$ is Objective Caml. Any general-purpose programming language with sufficiently well-defined semantics could be used instead. The technical requirement is that $\mathcal{P}$ forms a typed partial combinatory algebra [12].

[2]The *type* τ_A is not a set but just a formal expression in $\mathcal{P}$, which is why we distinguish between τ_A and its set of values $[\![\tau_A]\!]$.

```
Definition CommutativeGroup :=
thy
  Parameter s : Set.
  Parameter zero : s.
  Parameter add : s → s → s.
  Parameter neg : s → s.

  Implicit Type x y z : s.

  Definition sub x y := add x (neg y).

  Axiom add_associative: ∀ x y z, add x (add y z) = add (add x y) z.
  Axiom zero_neutral:    ∀ x, add x zero = x.
  Axiom neg_inverse:     ∀ x, add x (neg x) = zero.
  Axiom add_commutative: ∀ x y, add x y = add y x.
end.
```

Figure 1: The theory of a commutative group

RZ translates a theory to a *module specification*, which is an OCaml module type[3] (consisting of type declarations and definitions, and value declarations) annotated with assertions, written as comments, which state the properties that must be satisfied by the declared types and values. We demonstrate the translation procedure on a few typical examples.

A set declaration `Parameter` s : `Set` is translated to

```
type s
(** predicate (≈s) : s → s → bool *)
(** assertion symmetric_s :  ∀ x:s, y:s,  x ≈s y → y ≈s x
    assertion transitive_s :
      ∀ x:s, y:s, z:s,  x ≈s y ∧ y ≈s z → x ≈s z *)
(** predicate ‖s‖ : s → bool *)
(** assertion support_def_s :  ∀ x:s,  x : ‖s‖ ↔ x ≈s x *)
```

This says that the programmer should define a type `s`, and a relation $\approx_{\mathtt{s}}$ on `s` which is symmetric and transitive, in other words a per $(\mathtt{s}, \approx_{\mathtt{s}})$. The last two lines define the support $\|s\|$ discussed earlier, viewed as a predicate rather than a subset. Note that assertions are written inside comments, which is necessary as OCaml does not know about assertions. Another important observation is that $\approx_{\mathtt{s}}$ and $\|s\|$ are abstract predicates which are *not* required to be computable. We cannot expect $\approx_{\mathtt{s}}$ to be computable in general, e.g., when `s` implements a group with an undecidable word problem [16].

A value declaration `Parameter zero` : `s` is translated to

```
val zero : s
(**  assertion zero_support :  zero : ‖s‖ *)
```

which says that the programmer should define a value `zero` of type `s` which is in the support $\|\mathtt{s}\|$.

The definition of `sub` in Figure 1 is translated to

[3]Module types are also called *signatures* and vaguely correspond to header files in C, interfaces or abstract classes in Java, pure virtual classes in C++, and declarations in Haskell.

```
val sub : s → s → s
(**  assertion sub_def :
       sub ≈s → s → s (fun x : s ⇒ fun y : s ⇒ add x (neg y)) *)
```

The above assertion does *not* force `sub` x y to be implemented as `add` x (`neg` y), only to be equivalent to it with respect to the per. This is useful, as often the easiest way to define a value is not the most efficient way to compute it. The programmer is not limited to a purely functional programming style and is free to implement a specification using any features that exist in OCaml, including computational effects such as state and exceptions.

The driving force behind the translation of logic is a meta-theorem [19, 4.4.10] saying that under the realizability interpretation every formula ϕ is equivalent to one that says, informally speaking, "there exists $u \in |\phi|$, such that $u \Vdash \phi$", where $|\phi|$ is a type computed from ϕ and $u \Vdash \phi$ stands for "u realizes ϕ". Furthermore, the formula $u \Vdash \phi$ is negative, meaning that it may contain $\land$, $\to$, $\forall$, $=$, $\neg$, $\bot$, $\top$, but not $\lor$ or $\exists$. A welcome consequence of this is that the interpretation of $u \Vdash \phi$ is the same under both the constructive and classical reading. Therefore, programmers are able to understand the translation even if they are not familiar with constructive logic (which usually they are not).

The translation of a predicate ϕ then consists of its underlying type $|\phi|$ of realizers and the relation $u \Vdash \phi$, expressed as a negative formula. Thus an axiom `Axiom A` : ϕ in the input is translated to

```
val u : |φ|
(** assertion A : u ⊩ φ *)
```

which requires the programmer to validate ϕ by providing a realizer for it. The axioms of a commutative group are universally quantified equations, which are negative formulas. As such they have no computational content (the underlying type of realizers is `unit`) and RZ translates them directly to assertions, e.g., commutativity is translated as

```
(**  assertion add_commutative :
       ∀ (x:‖s‖, y:‖s‖),  add x y ≈s add y x *)
```

To get an interesting example, suppose we have already defined the usual structure of complex numbers `complex` and consider the "axiom" stating that every complex number has a square root:

```
Axiom sqrt : ∀ z : complex, ∃ w : complex, z = mul w w.
```

The translation is

```
val sqrt : complex → complex
(**  assertion sqrt : ∀ (z:‖complex‖),
       let p = sqrt z in p : ‖complex‖ ∧ z ≈complex mul p p *)
```

The axiom is validated by a value `sqrt` which computes square roots. Crucially, `sqrt` is *not* required to be extensional, i.e., it may compute two different square roots from two different representatives for the same complex number. In the language of Type Two Effectivity we would say that `sqrt` realizes a multi-valued function.

To see that the logic of RZ *must* be constructive, assume `nat` denotes the set of natural numbers, and consider the classically valid statement[4]

```
Axiom lpo : ∀ f : nat → nat,
  [‘zero: ∀ n : nat, f n = zero] ∨
  [‘nonzero: ¬ (∀ n : nat, f n = zero)].
```

is translated to the specification

```
val lpo : (nat → nat) → [‘zero | ‘nonzero]
(** assertion lpo : ∀ (f:||nat → nat||),
      (match lpo f with
         ‘zero ⇒ ∀ (n:||nat||),  f n ≈nat zero
       | ‘nonzero ⇒ ¬ (∀ (n:||nat||),  f n ≈nat zero)) *)
```

In order to validate the axiom, we would have to implement a function `lpo` which accepts as input (a representative of) a function $f : \mathbb{N} \to \mathbb{N}$ and outputs either ‘`zero` or ‘`nonzero`, depending on whether f is constantly zero or not. But the existence of such a decision procedure is equivalent to the existence of a Halting Oracle.

3 Implementing Real Numbers

There are several ways to characterize or construct real numbers. Even though they all result in isomorphic structures (as ordered fields), the choice of a representation and basic operations can have an enormous effect on efficiency of an implementation. Since we wanted to achieve performance that was comparable to fast implementations of exact real arithmetic such as iRRAM [15, 14], RealLib [10, 9] and MPFR [8], we looked for a theoretical model that would correspond closely to these under translation by RZ. A good starting point are the following observations about characteristics of iRRAM, RealLib and MPFR:

1. They are based on fast large integer libraries, such as GMP [1].
2. They work with *dyadic* rationals (those whose denominator is a power of two) rather than arbitrary ones.
3. On top of dyadic rationals, they implement interval arithmetic [13] and use intervals as approximations to reals.
4. Computations are started at a certain initial precision. As errors propagate, the quality of results deteriorates. If the final result is not precise enough, the entire computation is restarted from scratch with better initial precision.

It may seem wasteful to restart entire computations from scratch when the initial precision turns out to be too low. Indeed, earlier implementations of exact reals worked by propagating the precision backwards through intermediate computations in order to guarantee a final result with goal precision. But this often turned out to be even more expensive because the needed intermediate precisions tend to be overestimated so that too much work is done.

[4]In RZ the disjuncts ϕ, ψ in a disjunction $\phi \vee \psi$ may be labelled as $[‘\ell_1 : \phi] \vee [‘\ell_2 : \psi]$ for easier reference.

Prohibitive memory consumption is another kind of problem that may occur when the representation of reals contains computation trees. For example, if a number is computed as a sum $\sum_{k=0}^{n} f(k)$ its computation tree has size $\Theta(n)$, which may cause problems for large values of n. The iRRAM and MPFR avoid storing computation trees altogether (but take control of the main loop of the program), while RealLib stores computation trees implicitly, for example by storing f instead of the computation tree corresponding to the sum.

With all these issues in mind we looked for an axiomatizations of integers, dyadics, intervals, and reals that would give us suitable specifications.

3.1 Integers

Integers $\mathbb{Z}$ may be described concisely as the initial ring. Such a "mathematically optimal" characterization is not suitable for implementation, because it forces us to implement *everything*, even equality testing and linear ordering of $\mathbb{Z}$, in terms of unique ring homomorphisms from $\mathbb{Z}$. Instead an RZ theory of integers should mention those operations and properties that are actually computationally useful, even if some of them are interderivable. This allows the programmer to implement them all as efficiently as possible.

We used the axiomatization of integers shown in Appendix B. The `include` statement incorporates the theory of a decidable ordered ring from Appendix A. We define the natural numbers `nat` as a subset of integers, and state the usual induction principle, which RZ translates as a specification for a polymorphic function

```
val induction : α → (nat → α → α) → nat → α
```

with the following assertion: given any type α, an (abstract) predicate p : `nat` $\to \alpha \to$ `bool`, an element $x : \alpha$, and a function f : `nat` $\to \alpha \to \alpha$ such that $p\ 0\ x$ and, for all n : `nat` and $y : \alpha$, $p\ n\ y$ implies $p\ (n+1)(f\ n\ y)$, then $p\ n$ (`induction` $x\ f\ n$) for all n : `nat`. A moment's thought reveals that the assertion can be satisfied if we define `induction` to be the recursion operator characterized by

$$\begin{aligned} \texttt{induction}\, x\ f\ 0 &= x\ , \\ \texttt{induction}\, x\ f\ (n+1) &= f\ n\ (\texttt{induction}\, x\ f\ n)\ . \end{aligned}$$

The fact that the nonnegative integers satisfy the induction principle determines the ring of integers uniquely up to isomorphism. The rest of the axiomatization of integers deals with quotients and powers of two. This part of the theory is not strictly necessary, but is useful for an implementation of dyadic rationals.

For the implementation of integer arithmetic we used the *Numerix* library by Michel Quercia [17]. We also tested our implementation with GMP [1]. Both Numerix and GMP give similar performance which is much better than that of `Big_int` module from Objective Caml standard library.

3.2 Dyadic Rationals

A dyadic rational has the form $m \cdot 2^{-k}$ where $m \in \mathbb{Z}$ is the mantissa and $k \in \mathbb{N}$ is the exponent. The dyadic rationals are more efficient than the ordinary ones both in terms of memory consumption and basic arithmetic operations $+$, $-$

and $\times$. The fact that dyadic rationals form a decidable ordered ring, rather than a field, does not present a problem, because we still have *approximate division*: for all x and $y > 0$ we can find z such that $z \cdot y$ is as close to x as we wish. In fact, even though exact ring operations on dyadics are available, interval arithmetic uses their approximate versions in order to reduce memory consumption. Thus our axiomatization of dyadic rationals in Appendix B states not only that dyadics form a decidable ordered ring, but also that the basic operations may be approximated from below and above. For example, the axiom `add_approx_down` says that for all x and y there exists z such that $z \leq x + y \leq z + 2^{-k}$. The axiom is valid since we could just take $z = x + y$, but that misses the point. An efficient realizer for the axiom would compute z as the sum $x' + y'$ where x' and y' are suitably rounded x and y, so that they have smaller mantissas and exponents.

A careful inspection of theory `Dyadic` reveals that no axiom requires every element of the ring to be of the form $m \cdot 2^{-k}$. Indeed, we could take any decidable ordered ring in which the dyadic rationals are dense. This is quite similar to the *ring of base reals* in Exact Geometric Computation [21]. Note also that the axiom `magnitude`, which states that for every x there exists an integer k such that $2^k \leq |x| < 2^{k+1}$, implies that the ring is Archimedean.

3.3 Dyadic Intervals

Our intention is to approximate real numbers with intervals $[a, b]$ whose endpoints are dyadic rationals, or *dyadic intervals* for short.[5] It is convenient to adjoin an element `undefined`, corresponding to the interval $(-\infty, \infty)$, which allows for undefined results like division by zero. Dyadic intervals form a decidable conditional upper semilattice (cusl) under ordering by reverse inclusion, with `undefined` the least element.

Instead of following Moore's [13] definitions of addition, subtraction and multiplication of intervals, we make an adjustment that sacrifices a little bit of precision for quite a bit of speed and memory. The idea is to represent intervals as balls $[c - r, c + r]$, and then to approximate an interval by a slightly larger one whose radius r has a small mantissa. The approximation may save almost half the memory, as well as improve performance of basic operations. However, this means that Moore's exact interval operations become approximate. This is an acceptable compromise because typically we do not care about exact widths of intervals.

Our axiomatization of interval arithmetic is the theory `DyadicInterval` shown in Appendix D. First we include the theory of a conditional upper semilattice, see Appendix C. The operation `make` l u constructs an interval $[l, u]$. Note that the type of `make` is dependent, since u is required to be larger than l. RZ removes dependent types in the translation but still outputs obligations that make sure `make` is never used illegally. The operations `lower` and `upper` compute the endpoints of an interval, as witnessed by the axiom `endpoints`. Axiom `below_is_superset` characterizes the partial order as superset.

We axiomatize interval approximation discussed earlier by stipulating a function `normalize` for which there exists a tolerance factor $t \geq 1$ such that,

[5]Strictly speaking, since the real numbers are constructed from such intervals, a dyadic interval *is* the pair $[a, b]$ of its endpoints, and not the set of reals between a and b.

$[c' - r', c' + r'] =$ `normalize` $[c - r, c + r] \supseteq [c - r, c + r]$ and $r' \le t \cdot r$. We could trivially implement `normalize` as identity and $t = 1$, but the intention is to take $t = 1 + 2^{-b}$ for a fixed b, $c' = c$, and r' with at most b-bit mantissa and slightly larger than r.

Given a function $f : \mathsf{D} \to \mathsf{D}$ on the set of dyadic rationals D, say that $[l', u']$ *contains the f-image* of $[l, u]$, written as $f[l, u] \subseteq [l', u']$, if, for all $x \in \mathsf{D}$, $x \in [l, u]$ implies $f(x) \in [l', u']$. An *approximate f-image* of $[l, u]$ is an interval $[l', u']$ such that $f[l, u] \subseteq [l', u']$, and whenever $f[l, u] \subseteq [l'', u'']$ then $[l', u'] \subseteq$ `normalize` $[l'', u'']$. We similarly define an approximate image of a binary function $\mathsf{D} \times \mathsf{D} \to \mathsf{D}$.

The axioms for $+$, $-$, $\times$, min and max on intervals state that the corresponding operations on dyadic rationals have approximate images, and the RZ translation asks for operations which compute them. One possibility is to use Moore's interval arithmetic which computes exact images, but we implemented operations with normalization of intervals built in. The axiom for negation requires an exact image because nothing is gained by computing an approximate one. The axiom for division follows the approximate image idea but must be expressed differently because dyadic rationals do not form a field.

The last part of the theory `DyadicInterval` introduces a decidable preorder `less` on intervals. This is used in the construction of real numbers for linear ordering of reals.

3.4 Interval Domain

One of the characteristics of real numbers is that arbitrarily close to a real we can find a (dyadic) rational, i.e.,[6]

```
Axiom dense: ∀ x : real, ∀ k : nat, ∃ d : dyadic,
                 abs (x - incl d) ≤ incl (pow2 (neg k)).
```

This translates to the specification

```
val dense : real → nat → dyadic
(** assertion dense :
    ∀ (x:‖real‖, k:‖nat‖), let p = dense x k in
      p : ‖dyadic‖ ∧ abs (x - incl p)≤ incl (pow2 (neg k)) *)
```

which says that `dense` $x\,k$ computes a dyadic approximation within 2^{-k} of x. Such a function is of course essential for any implementation of exact reals, but taking it as one of the *basic* operations from which others are constructed leads to the sort of undesirable implementation that is based on backward propagation of accuracy goals. We have to look for an axiomatization of reals in which density of rationals comes as an afterthought.

As mentioned earlier, efficient implementations of exact real arithmetic compute in stages. We can represent this by a function

$$\texttt{stage} : \texttt{real} \to \texttt{nat} \to \texttt{interval}\ ,$$

where `stage` x k is the approximation to x obtained at stage k. This is very similar to `dense` above, except that there is no guarantee about the quality

[6]We assume that `incl` is the inclusion of dyadics into reals and that `pow2` (`neg` `k`) stands for 2^{-k}.

of k-th stage. Instead, all we know is that the next stage is no worse than the previous one[7] and that x is the only number which is approximated by all stages. In other words, x is the supremum of the chain of its approximations, which strongly suggests that we should look at a domain-theoretic construction of real numbers.

The poset ID of dyadic intervals may be completed to Dana Scott's *interval domain* IR [18], which is the set of closed real intervals ordered by reverse inclusion $\supseteq$. The completion $i : \mathsf{ID} \to \mathsf{IR}$ is a universal continuous map to ω-cpos,[8] i.e., for every continuous $f : \mathsf{ID} \to C$ to an ω-cpo C there exists a unique continuous $g : \mathsf{IR} \to C$ such that $f = g \circ i$. This characterizes IR up to isomorphism but is a bit inconvenient to write down in RZ.[9] Thus we use another characterization, namely that $i : \mathsf{ID} \to \mathsf{IR}$ is a continuous embedding to an ω-cpo with dense image, in the sense that every element of IR is the supremum of a chain from ID.

If we required that $i : \mathsf{ID} \to \mathsf{IR}$ be a universal *monotone* map, i.e., such that every monotone $f : \mathsf{ID} \to C$ has a unique continuous extension $\mathsf{IR} \to C$ along i, we would obtain an alternative domain-theoretic model of real numbers known as *the algebraic interval domain.* For a while we hesitated about which of the two models we should adopt, until we discovered that, although mathematically different, both structures can be implemented with exactly the same data structures and functions. This curious fact is explained when we notice that in both cases the elements of the completion are represented by chains of dyadic intervals, but the chains represent different things. In one case they represent their intersections (which are real intervals), while in the other they represent ideals. In the end we chose the interval domain IR because it corresponds directly to real interval arithmetic.

An RZ axiomatization of the relevant order-theoretic structures is shown in Appendix C:

`Poset` axiomatizes the theory of a poset. It also defines basic concepts such as maximality, chain, upper bound and supremum.

`DecidableCusl` axiomatizes a conditional upper semilattice with a decidable order. This theory is used in the axiomatization of dyadic intervals.

`CompletePoset` axiomatizes an ω-cpo. It is just the theory of a poset with an additional axiom stating that every chain has a supremum. RZ translates it to a specification for an operator computing suprema of chains.

`ChainCompletion` describes the ω-chain completion of a poset P. The axiom `stage` states that every element of the completion is the supremum of a chain in P. It is translated to a function `stage` discussed earlier.

The theory `RealInterval` describing the interval domain IR is just the parameterized theory `ChainCompletion` applied to `DyadicInterval`, see Appendix D.

[7]Actually, iRRAM does not guarantee that approximations form a nested chain of intervals, but this is not essential for our argument, and can be dealt with anyhow.

[8]Recall that an *ω-cpo* is a poset in which every increasing sequence (a chain) has a supremum, and that a monotone map between posets is *continuous* if it preserves existing suprema of chains.

[9]While RZ is well suited for descriptions of objects of a category, dealing with morphisms is cumbersome. There is definitely room for improvement here.

3.5 Real Numbers

We finally come to the theory of real numbers, shown in Appendix E. We first include the theory `OrderedField`, see Appendix A, which takes care of basic arithmetic and the lattice operations min, max. The rest of the axiomatization deals with the relationship between reals and the interval domain, continuity of arithmetic operations, completeness properties of reals, linear order, and the Archimedean property. We briefly comment on each of these.

Reals and the interval domain. The reals are isomorphic to the space of maximal elements of the interval domain. Thus we postulate two maps `to_interval` and `of_interval` which convert real numbers to maximal intervals and vice versa. Because in the implementation we happen to use the same datatype to represent both reals and intervals, the conversions are just identities. In fact, we could have avoided them altogether if we defined reals to be the maximal intervals, but we did not do that because we wanted to keep a clear distinction between the abstract characterization of reals and their representation as maximal intervals.

Continuity of arithmetic operations. Our implementation represents real numbers (and real intervals) as chains of dyadic intervals converging to them. Thus to compute $f(x)$ for a continuous $f : \mathbb{R} \to \mathbb{R}$ and $x \in \mathbb{R}$ represented by a chain $d_0 \leq d_1 \leq \cdots$ in ID, we need to find a chain $e_0 \leq e_1 \leq \cdots$ in ID whose supremum is $f(x)$. Ideally, e_i should depend only on d_i because that allows us to compute the i-th approximation of the result by computing only the i-th approximation of the argument. What we are asking for is a particular form of continuity of f, namely that there exist a continuous $g : \mathsf{ID} \to \mathsf{ID}$ such that $f(\sup_i d_i) = \sup_i(g(d_i))$ for every chain $(d_i)_i$ whose supremum is in $\mathbb{R}$. In this case we can take $e_i = g(d_i)$. We defer a general discussion about this kind of continuity for general functions to another occasion, and just observe that the basic arithmetic operations are indeed just extensions of corresponding (approximate) operations on dyadic intervals, as is stated in the theory `Real`.

Completeness. We express completeness of reals with a variant of Cauchy completeness. Say that a sequence $(a_i)_i$ is *Cauchy* if there exists a decreasing sequence $r_0 \geq r_1 \geq r_2 \geq \cdots$ of non-negative numbers whose infimum is 0 and such that $|a_m - a_k| \leq r_k$ whenever $k \leq m$. We call r the *remainder sequence* because it tells us how far from the limit the terms of the sequence are. Further, say that x is an *accumulation point* of $(a_i)_i$ if for every k the infimum of the sequence $(|x - a_i|)_{i \geq k}$ is zero. Now completeness of $\mathbb{R}$ means that every Cauchy sequence of reals has an accumulation point (which turns out to be unique). However, we still have a choice as to what it means for a sequence of non-negative numbers $(a_i)_i$ to have infimum zero:

1. for every $\epsilon > 0$ there exists k such that $a_i \leq \epsilon$ for all $i \geq k$, or
2. if $x \leq a_i$ for all i then $x \leq 0$.

The first definition is the usual constructive one, but we use the second one because it is better suited for the order-theoretic approach we have taken. The axiom `lim`, which states that every Cauchy sequence has an accumulation point, translates to a specification for a function

```
val lim : (I.nat → s) → (I.nat → s) → s
```

which computes the limit from a sequence and its remainder sequence. Had we taken the first definition of infimum above, `lim` would also require as input a realizer telling us how fast the reminder sequence decreases to zero. This is something we want to avoid, as it can be quite cumbersome to compute such information, and the implementation of `lim` does not need it anyway.

Linear order. The inclusion of `OrderedField` into the theory of reals axiomatizes the lattice structure in terms of the operations `min` and `max`. The order relation $x \leq y$ is defined as max x $y = y$, and the strict order $<$ is defined as the negation of $\leq$. From this RZ determines that $\leq$ and $<$ are stable and do not carry any computational content. However, we can do better and postulate *partial* comparison which takes values in the domain of partial booleans `PartialBoolean`, see Appendix C. The axiom `less` states that for all x and y there exists a partial boolean b which is true when $x < y$ and false when $x > y$. The axiom does not state what the value of b should be when $x = y$, but it can be easily seen that the only (computable) option is the undefined value. RZ translates the axiom into a specification for a map `less` computing a partial boolean b from x and y. Such a b is represented by a chain of values ‘`undefined` that may eventually become either ‘`ff` or ‘`tt`. At stage k, `less` compares the k-th approximations of x and y with the help of `cmp_less` from `DyadicInterval` to see if it can reach a decision about the ordering of x and y.

Archimedean property. The last axiom `approx_to` asserts the Archimedean property of reals, namely that dyadic rationals form a dense subset of the reals. The translation is a specification for a function `approx_to` which accepts a real x and a natural number k, and computes a dyadic 2^{-k}-approximation of x. This can be implemented, and the axiom validated, by a search procedure which computes the stages of x until it finds an approximation whose width is no wider than 2^{-k}. We employ Markov's principle to prove that the search terminates: since it is impossible for all approximations of x to be wider than 2^{-k}, there is one which is no wider than 2^{-k}.

4 Discussion

Implementation and performance. The entire axiomatization consists of about 600 lines of code describing rings, ordered rings and fields, posets, cusls, poset completions, interval arithmetic and real numbers. The Objective Caml implementation reaches about 800 lines of code and includes modules for interfacing with a large integer library, dyadic rationals, dyadic and real intervals, partial booleans and real number.

We called our implementation of reals *Era*, which stands for Exact Real Arithmetic. We hoped, but did not expect the performance of Era to rival that of other libraries. Indeed, initial measurements show that iRRAM is about 40 times faster than Era. Such a large difference can be partially explained by the fact that C++ generally compiles to more efficient code than Objective

Caml, and that iRRAM is a much more mature and highly optimized piece of software. The gap ought to decrease in the future, as we find ways to improve the performance of Era.

Nevertheless, we have achieved our main goal, namely to demonstrate that there is a harmony between theory and practice. Era is a *direct* implementation of a standard domain-theoretic construction of reals as the maximal elements of interval domain which uses data structures and algorithms that closely match those of iRRAM and RealLib: computations on reals are performed in stages, where each stage computes with dyadic intervals and is entirely independent of previous stages. Era does not suffer from unacceptable memory consumption because it does not store extremely large computation trees directly. Instead, a possibly large computation such as a sum is stored as a closure which generates the computation tree on demand.

A lesson for theoreticians. The interplay between theory and practice contains an interesting lesson about how practitioners take advantage of validity of extra-logical axioms. The first such example is Markov principle, which states that under suitable circumstances missing information can be recovered. In our case this applies to chains of intervals converging to a real: we need not store explicit information about the speed of convergence, because it may be recovered by computing successive terms of a chain until they become sufficiently precise. This is a welcome optimization that helps save both space and time. Note however that actually applying Markov principle and searching for good enough an approximation is expensive and undesirable. So we avoid doing this, except at the top level when a final result must be computed to a desired precision.

The second extra-logical axiom which practitioners essentially rely upon is a continuity principle stating that, in a suitable sense, "all (computable) functions are continuous". Among other things this implies that a modulus-of-continuity functional is computable. However, in practice such a functional is horribly inefficient and should never be used. Instead we use the principle to our advantage by representing functions so that their continuity is explicitly exposed, which then allows us to compute more efficiently. Specifically, we represent real functions as mappings from dyadic intervals to dyadic intervals, which allows for efficient computation by independent stages of approximation. The continuity principle guarantees that any function of interest can be so represented.

The lesson for theoreticians to observe here is that Markov principle and continuity principle are never used explicitly in a computation, but rather *implicitly* in the choice of data structures. Thus Markov principle allows us to remove information about the speed of convergence from the representation of a real number, while the continuity principle allows us to augment the representation of a real function with information about its action on intervals. Therefore, in order to aid practical implementations, we should develop constructive and computable mathematics which avoids explicit uses of extra-logical axioms, but does consider the possibility of them being true when mathematical structures are defined and constructed.

RZ and Coq. RZ is similar to various tools for formalization of mathematics, most notably to the proof assistant Coq [6], which not only allows one to axiomatize theories, but also to construct models and formally prove their properties. Coq is able to extract trusted code from proofs, which gives us a specification as well as its implementation. While this has turned out to be a very successful technique in many respects, current code-extraction methods produce only purely functional code which does not compare favorably to efficient hand-crafted code at all. The goal of RZ was to give programmers a light-weight tool which would allow them to connect the theoretical models with implementations, but would not force them to write proofs instead of programs.

In fact, one could use RZ to axiomatize theories and then proceed with their implementation within Coq. In this case RZ can be seen as a tool which automatically separates the computationally relevant and irrelevant parts of a theory,[10] something that is done by hand in Coq. Another possibility would be to manually write code and then use a proof assistant such as Coq, to prove that the code actually satisfies the RZ specification.

Future directions. The current Era implementation is only an initial prototype which we intend to extend and improve. The Objective Caml module system allows us to easily experiment with various libraries for big integers and interval arithmetic, as well as to mix floating point computations with exact ones. These are all possibilities we wish to explore.

The realizability model on which RZ is based allows us to use computational effects in the implementation (which we do, for example to cache the current approximation of a real), but it does not allow effects to be exposed at the level of logic. Thus we cannot reasonably axiomatize an operation which changes a real number in-place. We would like to extend the input language in such a way that it would allow us to express exceptions, mutable values, and possibly other effects.

Lastly, we observe that practical implementations lag behind theoretical developments in constructive and computable mathematics, as they usually deal just with numbers and functions on them. In the future we would like to see data structures for manifolds, Hilbert spaces, vector analysis, etc. Hopefully, tools like like RZ will help manage the complexity of the task.

References

[1] *GNU Multiple Precision Arithmetic Library.* `http://gmplib.org/`.

[2] Andrej Bauer. *The Realizability Approach to Computable Analysis and Topology.* PhD thesis, Carnegie Mellon University, 2000.

[3] Andrej Bauer. Realizability as the connection between constructive and computable mathematics. Available at `http://math.andrej.com`, 2005. Tutorial lecture at Computability and Complexity in Analysis 2005, Kyoto, Japan.

[4] Andrej Bauer and Christopher Stone. RZ. `http://math.andrej.com/rz/`.

[5] Andrej Bauer and Christopher Stone. RZ: a tool for bringing constructive and computable mathematics closer to programming practice. In *Computability in Europe 2007*, June 2007.

[10]These are the kinds `Set` and `Prop` in Coq, respectively

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[7] Vasco Brattka, Matthias Schröder, Klaus Weihrauch, and Ning Zhong. Computability and complexity in analysis. Informatik Berichte 302, FernUniversität in Hagen, Hagen, August 2003. Proccedings, International Conference, CCA 2003, Cincinnati, USA, August 28–30, 2003.

[8] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. *The MPFR Library.* INRIA. `http://www.mpfr.org/`.

[9] Branimir Lambov. *The RealLib Project.* BRICS, University of Aarhus. `http://www.brics.dk/ barnie/RealLib/`.

[10] Branimir Lambov. RealLib: An efficient implementation of exact real arithmetic. *Mathematical Structures in Computer Science*, 17(1):81–98, 2007.

[11] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, documentation and user's manual - release 3.08. Technical report, INRIA, July 2004.

[12] John Longley. Matching typed and untyped realizability. *Electr. Notes Theor. Comput. Sci.*, 23(1), 1999.

[13] Ramon Moore. *Interval Analysis.* Automatic Computation. Prentice Hall, 1966.

[14] Norbert Müller. *iRRAM – Exact arithmetic in C++.* Universität Trier. `http://www.informatik.uni-trier.de/iRRAM/`.

[15] Norbert Th. Müller. Towards a real Real RAM: a prototype using C++. In Ker-I Ko, Norbert Müller, and Klaus Weihrauch, editors, *Computability and Complexity in Analysis*, pages 59–66. Universität Trier, 1996. Second CCA Workshop, Trier, August 22–23, 1996.

[16] E. Post. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic*, 12:1–11, 1947.

[17] Michel Quercia. *Numerix: Big Integer Library, version 0.22.* INRIA. `http://pauillac.inria.fr/~quercia/`.

[18] D.S. Scott. Lattice theory, datatypes and semantics. In *Formal semantics of programming languages*, pages 66–106. Prentice-Hall, 1972.

[19] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction, Vol. 1.* Number 121 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

[20] Klaus Weihrauch. *Computable Analysis.* Springer, Berlin, 2000.

[21] Chee K. Yap. Theory of real computation according to EGC, 2006. To appear in LNCS Volume based on the Dagstuhl Seminar "Reliable Implementation of Real Number Algorithms: Theory and Practice", Jan 8-13, 2006.

A Rings and Fields

```
Definition CommutativeGroup :=
thy
  Parameter s : Set.
  Parameter zero : s.
  Parameter add : s → s → s.
  Parameter neg : s → s.
  Definition sub (x y : s) := add x (neg y).
  Implicit Type x y z : s.
  Axiom add_associative: ∀ x y z, add x (add y z) = add (add x y) z.
  Axiom add_commutative: ∀ x y, add x y = add y x.
  Axiom zero_neutral: ∀ x, add x zero = x.
  Axiom neg_inverse: ∀ x, add x (neg x) = zero.
end.

Definition CommutativeRingWithUnit :=
thy
  include CommutativeGroup.
  Implicit Type x y z : s.
  Parameter one : s.
  Parameter mul : s → s → s.
  Axiom nontrivial: ¬ (zero = one).
  Axiom mul_associative: ∀ x y z, mul (mul x y) z = mul x (mul y z).
  Axiom mul_commutative: ∀ x y, mul x y = mul y x.
  Axiom one_neutral: ∀ x, mul one x = x.
  Axiom distributive: ∀ x y z, mul x (add y z) = add (mul x y) (mul x z).
  Definition nonzero := { x | ¬ (x = zero) }.
end.

Definition OrderedRing :=
thy
  include CommutativeRingWithUnit.
  Implicit Type x y z : s.
  Parameter max : s → s → s.
  Definition min x y := neg (max (neg x) (neg y)).
  Axiom max_idempotent: ∀ x, max x x = x.
  Axiom max_commutative: ∀ x y, max x y = max y x.
  Axiom max_associative: ∀ x y z, max x (max y z) = max (max x y) z.
  Definition leq x y := (max x y = y).
  Definition lt x y := ¬ (leq y x).
  Definition positive := { x | lt zero x }.
  Axiom max_add: ∀ x y z, add (max x y) z = max (add x z) (add y z).
  Axiom max_mul: ∀ x y z, leq zero z → mul (max x y) z = max (mul x z) (mul y z).
  Axiom max_zero_one : max zero one = one.
  Definition abs x := max x (neg x).
  Definition dist x y := abs (sub x y).
end.

Definition DecidableOrderedRing :=
thy
  include OrderedRing.
  Implicit Type x y : s.
  Axiom cmp:  ∀ x y, [‘less : lt x y] ∨ [‘equal : x = y] ∨ [‘greater : lt y x].
  Axiom sgn:  ∀ x, [‘negative : lt x zero] ∨ [‘zero : x = zero] ∨ [‘positive : lt zero x].
  Axiom eq:   ∀ x y, [‘True : x = y] ∨ [‘False : ¬ (x = y)].
  Axiom neq:  ∀ x y, [‘False : x = y] ∨ [‘True : ¬ (x = y)].
end.

Definition OrderedField :=
thy
  include OrderedRing.
  Parameter inv : nonzero → nonzero.
  Axiom inv_inverse: ∀ x : nonzero, mul x (inv x) = one.
  Definition div (x : s) (y : nonzero) := mul x (inv y).
end.
```

B Integers and Dyadic Rationals

```
Definition Integer :=
thy
  include DecidableOrderedRing.
  Definition nat := { x : s | leq zero x }.
  Implicit Type x y z : s.
  Implicit Type k n : nat.
  Definition succ n := (add one n) : nat.
  Definition two := succ (succ zero).
  Axiom induction :
    ∀ M : thy Parameter p : nat → Prop. end,
      M.p zero → (∀ k, M.p k → M.p (succ k)) → ∀ k, M.p k.
  Definition quotient x (y : nonzero) :=
    the z, let t = sub x (mul z y) in leq zero t ∧ lt t (abs y).
  Parameter pow2 : nat → nat.
  Axiom pow2_is_power_of_two:
    pow2 zero = one ∧ ∀ k, pow2 (succ k) = mul two (pow2 k).
  Definition shift_right x (k : nat) := quotient x (pow2 k).
  Definition shift_left x (k : nat) := mul x (pow2 k).
  Axiom magnitude:
    ∀ x : nonzero, ∃ k : nat,
      leq (pow2 k) (abs x) ∧ lt (abs x) (pow2 (succ k)).
end.

Definition Dyadic (I : Integer) :=
thy
  include DecidableOrderedRing.
  Definition two := add one one.
  Implicit Type x y z w : s.
  Implicit Type k : I.nat.
  Parameter of_integer: I.s → s.
  Axiom of_integer_hom:
      of_integer I.zero = zero ∧ of_integer I.one = one ∧
      ∀ m n : I.s, (of_integer (I.add m n) = add (of_integer m) (of_integer n)).
  Parameter pow2: I.s → s.
  Axiom pow2_is_power_of_two:
    pow2 I.zero = one ∧ pow2 I.one = two ∧
    ∀ m n : I.s, pow2 (I.add m n) = mul (pow2 m) (pow2 n).
  Definition half := the x, mul x two = one.
  Definition halve x := mul x half.
  Axiom magnitude:
    ∀ x : nonzero, ∃ k : I.s,
      leq (pow2 k) (abs x) ∧ lt (abs x) (pow2 (I.add I.one k)).
  Definition up k x   := add x (pow2 (I.neg k)).
  Definition down k x := sub x (pow2 (I.neg k)).
  Definition approx_down k x y := leq x y ∧ leq y (up k x).
  Definition approx_up k x y := leq (down k x) y ∧ leq y x.
  Axiom add_approx_down: ∀ k x y, ∃ z, approx_down k z (add x y).
  Axiom add_approx_up:   ∀ k x y, ∃ z, approx_up   k z (add x y).
  Axiom sub_approx_down: ∀ k x y, ∃ z, approx_down k z (sub x y).
  Axiom sub_approx_up:   ∀ k x y, ∃ z, approx_up   k z (sub x y).
  Axiom mul_approx_down: ∀ k x y, ∃ z, approx_down k z (mul x y).
  Axiom mul_approx_up:   ∀ k x y, ∃ z, approx_up   k z (mul x y).
  Axiom div_approx_down:
    ∀ k x, ∀ y : positive, ∃ z, leq (mul y z) x ∧ leq x (mul y (up k z)).
  Axiom div_approx_up:
    ∀ k x, ∀ y : positive, ∃ z, leq (mul y (down k z)) x ∧ leq x (mul y z).
end.
```

C Posets

```
Definition Poset (I : Integer) :=
thy
  Parameter s : Set.
  Implicit Type x y z : s.
  Parameter below : s → s → Stable.
  Axiom below_reflexive : ∀ x, below x x.
  Axiom below_transitive : ∀ x y z, below x y ∧ below y z → below x z.
  Axiom below_antisymmetric : ∀ x y, below x y ∧ below y x → x = y.
  Definition maximal := { x | ∀ y, below x y → x = y }.
  Definition chain := { a : I.nat → s | ∀ k : I.nat, below (a k) (a (I.succ k)) }.
  Definition upper_bound (a : I.nat → s) x := ∀ k : I.nat, below (a k) x.
  Definition supremum (a : chain) (x : s) :=
    (upper_bound a x) ∧ (∀ y, upper_bound a y → below x y).
  Definition is_monotone (f : s → s) := ∀ x y, below x y → below (f x) (f y).
  Definition is_continuous (f : s → s) :=
    is_monotone f ∧
    ∀ x, ∀ a : chain, supremum a x → supremum (fun k : I.nat ⇒ f (a k)) (f x).
  Definition monotone := { f : s → s | is_monotone f }.
  Definition continuous := { f : s → s | is_continuous f }.
  Definition monotone2 :=
    { f : s → s → s | ∀ x, is_monotone (f x) ∧ is_monotone (fun y ⇒ f y x) }.
  Definition continuous2 :=
    { f : s → s → s | ∀ x, is_continuous (f x) ∧ is_continuous (fun y ⇒ f y x) }.
end.

Definition DecidableCusl (I : Integer):=
thy
  include Poset I.
  Implicit Type x y z : s.
  Parameter undefined : s.
  Axiom undefined_is_least: ∀ x, below undefined x.
  Axiom cmp_below :
    ∀ x y,
      [‘below : below x y] ∨ [‘above : below y x] ∨
      [‘incomparable : ¬ (below x y ∨ below y x)].
  Definition inconsistent x y := ∀ z, ¬ (below x z ∧ below y z).
  Definition consistent x y := ¬ (inconsistent x y).
  Axiom join :
    ∀ x y, consistent x y → ∃ z,
      below x z ∧ below y z ∧ (∀ w : s, below x w ∧ below y w → below z w).
end.

Definition CompletePoset (I : Integer):=
thy
  include Poset I.
  Axiom sup : ∀ a : chain, ∃ x : s, supremum a x.
end.

Definition ChainCompletion
  (I : Integer)
  (P : Poset I) :=
thy
  include CompletePoset I.
  Parameter incl : P.s → s.
  Definition incl_chain (a : P.chain) := (fun n : I.nat ⇒ incl (a n)) : chain.
  Axiom incl_injective : ∀ x y : P.s, incl x = incl y → x = y.
  Axiom incl_monotone : ∀ x y : P.s, P.below x y → below (incl x) (incl y).
  Axiom incl_continuous:
    ∀ x : P.s, ∀ a : P.chain, P.supremum a x → supremum (incl_chain a) (incl x).
  Axiom stage : ∀ x : s, ∃ a : P.chain, supremum (incl_chain a) x.
  Definition extend (f : P.continuous) :=
    the g : continuous, ∀ x : P.s, incl (f x) = g (incl x).
  Definition extend2 (f : P.continuous2) :=
    the g : continuous2, ∀ x y : P.s, incl (f x y) = g (incl x) (incl y).
  Axiom make : ∀ a : P.chain, ∃ x : s, supremum (incl_chain a) x.
end.

Definition PartialBoolean (I : Number.Integer) :=
thy
  include CompletePoset I.
  Parameter undefined tt ff : s.
  Axiom below_is_flat : ∀ x y : s, below x y ↔ (¬ (x = undefined) → x = y).
  Axiom decide : ∀ x : s, ¬ (x = undefined) → [‘tt : x = tt] ∨ [‘ff : x = ff].
end.
```

D Dyadic and Real Intervals

```
Definition DyadicInterval
  (I : Integer)
  (D : Dyadic I) :=
thy
  include DecidableCusl I.
  Definition interval := { x : s | ¬ (x = undefined) }.
  Implicit Type x y z : s.
  Implicit Type u v t : D.s.
  Parameter make : [l : D.s] → { u : D.s | D.lt l u } → interval.
  Parameter lower upper : interval → D.s.
  Axiom endpoints: ∀ u v, D.lt u v → lower (make u v) = u ∧ upper (make u v) = v.
  Definition ball (c : D.s) (r : D.positive) := make (D.sub c r) (D.add c r).
  Definition center (x : interval) := D.halve (D.add (upper x) (lower x)).
  Definition radius (x : interval) := D.halve (D.sub (upper x) (lower x)).
  Definition elem u x := ¬ (x = undefined) → D.leq (lower x) u ∧ D.leq u (upper x).
  Axiom below_is_superset:
    ∀ x y, below x y ↔ (∀ u : D.s, elem u y → elem u x).
  Parameter normalize : interval → interval.
  Axiom tolerance :
    ∃ t : D.positive, ∀ x : interval,
      below (normalize x) x ∧ D.leq (radius (normalize x)) (D.mul t (radius x)).
  Definition below_approx x y := ¬ (x = undefined) → below (normalize x) y.
  Definition contains_image z (f : D.s → D.s → D.s) x y :=
    ∀ u v : D.s, elem u x ∧ elem v y → elem (f u v) z.
  Definition image_approx z (f : D.s → D.s → D.s) x y :=
    contains_image z f x y ∧
    (∀ z' : s, contains_image z' f x y → below_approx z' z).
  Axiom add : ∀ x y, ∃ z, image_approx z D.add x y.
  Axiom sub : ∀ x y, ∃ z, image_approx z D.sub x y.
  Axiom mul : ∀ x y, ∃ z, image_approx z D.mul x y.
  Axiom neg : ∀ x, ∃ y, ∀ u, (elem u x ↔ elem (D.neg u) y).
  Axiom min : ∀ x y, ∃ z, image_approx z D.min x y.
  Axiom max : ∀ x y, ∃ z, image_approx z D.max x y.
  Definition scale (u : D.nonzero) (x : interval) :=
    let v = D.mul u (lower x) in let w = D.mul u (upper x) in
      make (D.min v w) (D.max v w).
  Definition contains_div z x y :=
    ¬ (z = undefined) →
      ∀ u, ∀ v : D.nonzero, elem u x ∧ elem v y → elem u (scale v z).
  Axiom div :
    ∀ x y, ∃ z,
      contains_div z x y ∧ (∀ z' : s, contains_div z' x y → below_approx z' z).
  Definition less x y := D.lt (upper x) (lower y).
  Axiom cmp_less :
    ∀ x y, [`less : less x y] ∨ [`greater : less y x] ∨
               [`incomparable : ¬ (less x y) ∧ ¬ (less y x)].
end.

Definition RealInterval
  (I : Integer)
  (D : Dyadic I)
  (ID : DyadicInterval I D) := ChainCompletion I ID.
```

E Real Numbers

```
Definition Real
  (I : Integer)
  (D : Dyadic I)
  (ID : DyadicInterval I D)
  (IR : RealInterval I D ID) :=
thy
  include OrderedField.
  Implicit Type x y z : s.
  Implicit Type k m n : I.nat.
  Parameter of_interval : IR.maximal → s.
  Parameter to_interval : s → IR.maximal.
  Axiom reals_maximal:
    (∀ x, of_interval (to_interval x) = x) ∧ ∀ u : IR.s, to_interval (of_interval u) = u.
  Axiom stage : ∀ x, ∃ a : ID.chain, IR.supremum (IR.incl_chain a) (to_interval x).
  Definition continuous (f : s → s) :=
    ∃ g : ID.continuous, ∀ x, to_interval (f x) = IR.extend g (to_interval x).
  Definition continuous2 (f : s → s → s) :=
    ∃ g : ID.continuous2, ∀ x y,
      to_interval (f x y) = IR.extend2 g (to_interval x) (to_interval y).
  Axiom add_interval : continuous2 add.
  Axiom neg_interval : continuous  neg.
  Axiom sub_interval : continuous2 sub.
  Axiom mul_interval : continuous2 mul.
  Axiom min_interval : continuous2 min.
  Axiom max_interval : continuous2 max.
  Axiom abs_interval : continuous  abs.
  Axiom dist_interval: continuous2 dist.
  Axiom div_interval :
    ∃ g : ID.continuous2, ∀ x, ∀ y : nonzero,
      to_interval (div x y) = IR.extend2 g (to_interval x) (to_interval y).
  Axiom inv_interval :
    ∃ g : ID.continuous, ∀ y : nonzero, to_interval (inv y) = IR.extend g (to_interval y).
  Parameter of_integer:
    ∃ f : I.s → s,
      f I.zero = zero ∧ f I.one = one ∧ ∀ x y : I.s, f (I.add x y) = add (f x) (f y).
  Parameter of_dyadic : D.s → s.
  Axiom of_dyadic_hom:
    of_dyadic D.zero = zero ∧ of_dyadic D.one = one ∧
    (∀ x y : D.s, of_dyadic (D.add x y) = add (of_dyadic x) (of_dyadic y)) ∧
    (∀ x y : D.s, of_dyadic (D.mul x y) = mul (of_dyadic x) (of_dyadic y)) ∧
    (∀ x y : D.s, of_dyadic (D.max x y) = max (of_dyadic x) (of_dyadic y)).
  Definition infimum (a : I.nat → s) x :=
    (∀ k, leq x (a k)) ∧ (∀ y, (∀ k, leq y (a k)) → leq y x).
  Definition is_cauchy (a : I.nat → s) :=
    ∃ r : I.nat → s,
      (∀ k, leq (r (I.succ k)) (r k)) ∧ (infimum r zero) ∧
      (∀ k m, I.leq k m → leq (dist (a k) (a m)) (r k)).
  Axiom lim :
    ∀ a : I.nat → s, is_cauchy a →
      ∃ x, ∀ k, infimum (fun m ⇒ dist (a (I.add k m)) x) zero.
  Axiom less :
    ∀ x y, ∃ b : B.s, (lt x y ↔ b = B.tt) ∧ (lt y x ↔ b = B.ff).
  Axiom approx :
    ∀ x, ∀ k, ∃ d : D.s, leq (dist x (of_dyadic d)) (of_dyadic (D.pow2 (I.neg k))).
end.
```