AN EFFICIENT MODEL OF CONSTRUCTIVE REAL NUMBERS IN COQ

DANKO ILIḰ

Contents

1. Introduction

The work before you is an experiment in implementing exact real arithmetic in the Coq proof assistant. Although such efforts have already done by others, see [1][2][3], this attempt has taken another theoretical starting point. Following Stolzenberg's approach from [4], we consider the real number system built as follows.

One considers a theory of closed intervals with rational endpoints which is an Archimedean ordered field and a lattice, and also a binary relation of 'consistency' on the intervals which holds when two intervals intersect. This relation is not transitive, and so is not an equality, but one may profit from thinking about it as a kind of equality.

Next, one considers *sets* of the above intervals equipped with the same algebraic structure, inherited directly from the previous one, with the exception that inheriting the consistency relation now gives us a real equality relation on the sets of intervals. One says that a set of intervals is 'fine' if it contains an interval of arbitrary small length, and that it is 'consistent' if every two contained intervals intersect. A fine and consistent set of intervals is a 'real number'.

One now reflects for a moment and sees no problem in defining a theory of intervals with *'real'* endpoints by copying the developed theory of intervals with rational endpoints, and analogously defining a theory of *sets* of intervals with *'real'* endpoints. One defines for the fine and consistent sets of intervals with 'real' endpoints a limit functor and shows that the theory from the third paragraph above is complete, that it contains all of its limits, and so that a 'real number' is really a real number.

What is the advantage of this approach in relation the the approaches based on Cauchy sequence, or Dedekind cut, representation of real numbers? The author is not really sure. He can only express the feeling, drawn from the experience in defining Euler's number in it, that working with this framework does not pose heavy demands on using it in concrete situations.

When implementing a theory like this one, one should not be concerned solely with the theoretical model. In practice, having a good and fast library implementing integer and rational number arithmetic, has greater importance. For example, the rational arithmetic library from the standard library of Coq 8.1, although fast, is not completely complete, for it is missing, for example, the definition of min and max and their properties. We shall remark more on such points when time comes.

The notation for writing mathematics in this report will be the type theoretic notation of Coq. This is so because here we speak about implementations and this requires formality. The reader is advised to read the introduction chapter of the book [4] for a (better and) purely mathematical treatment.

2. Real numbers

2.1. **Theory of intervals.** An interval with rational endpoints is given by a left and right bound and a proof that left is left of right:

```
Record QI : Set :=
  QImake { QIleft     : Q;
           QIright    : Q;
           QInonempty : (QIleft<=QIright)%Q }.
```

2.1.1. *Order, consistency.* The less-than relation on this set determines whether the first operand (interval) is completelly left of the second and is defined as:

```
Definition QIlt (I J:QI) := (QIright I) < (QIleft J).
```

We want to have another relation, consistency, which determines whether two intervals intersect. It is defined like this:

```
Definition QIc (I J:QI) :=
  (Qmax (QIleft I) (QIleft J)) <= (Qmin (QIright I) (QIright J)).
```

Already at this stage we encounter a problem with the Coq standard library, as Qmax and Qmin are not defined there, and so we have to develop a few lemmas about them ourselves.

The consistency relation plays the role of an equality very often, like one can see from the following lemmas:

```
Lemma QIlt_trichotomy : forall I J:QI, (I<J) \/ (I~J) \/ (J<I).
Definition QIlc (I J:QI) := I<J \/ I~J.
Infix "<~" := QIlc (at level 70) : QI_scope.
```

```
Lemma QIlc_not_gt : forall I J:QI, (I<~J) <-> not (I>J).
Lemma QIc_sym : forall I J:QI, I~J -> J~I.
Lemma QIc_lc : forall I J:QI, I~J <-> I<~J /\ J<~I.
Lemma QIlt_lc_lt : forall I J K L:QI, I<J -> J<~K -> K<L -> I<L.
```

The proofs of all these go by unfolding the definitions and applying the lemmas about Qmax and Qmin. The reader himself can run the proofs if he is interested in the details.

2.1.2. *Superemum, infimum.* The following least upper bound and gretest lower bound functions create the latice structure on the intervals:

```
Definition QImax (I J:QI) := QImake
  (Qmax (QIleft I) (QIleft J)) (Qmax (QIright I) (QIright J))
  (QImax_nonempty I J) : QI.
Definition QImin (I J:QI) := QImake
  (Qmin (QIleft I) (QIleft J)) (Qmin (QIright I) (QIright J))
  (QImin_nonempty I J) : QI.
```

We have these lemmas:

```
Lemma QImax_el : forall I J:QI, forall x y:Q,
  QIel x I -> QIel y J -> QIel (Qmax x y) (QImax I J).
Lemma QImin_el : forall I J:QI, forall x y:Q,
  QIel x I -> QIel y J -> QIel (Qmin x y) (QImin I J).
Lemma QImax_lt : forall I J K:QI, (QImax I J)<K <-> I<K /\ J<K.
Lemma QIlt_max : forall I J K:QI, K<(QImax I J) <-> K<I \/ K<J.
```

And we should also, following Stolzenberg, have these:

```
forall I J:QI, (QImax I J) ~ I <-> (QImin I J) ~ J <-> I<~J.
forall I J:QI, QIsubset (QImax I J) I \/ QIsubset (QImax I J) J.
```

But,they are not there yet, mostly because they were not needed in the development of the library yet.

2.1.3. *Arithmetic.*

```
Definition QIsum (I J:QI) :=
  QImake (QIleft I + QIleft J) (QIright I + QIright J)
  (Qplus_le_compat _ _ _ _ (QInonempty I) (QInonempty J)) : QI.
Definition QIminus (I:QI) := QImake (- QIright I) (- QIleft I)
  (Qopp_le_compat _ _ (QInonempty I)) : QI.
Definition QIdifference (I J:QI) := QIsum I (QIminus J) : QI.
Definition QIproduct (I J:QI) :=
  let r:=QIleft I in
  let s:=QIright I in
  let u:=QIleft J in
  let v:=QIright J in
    QImake
    (Qmin (Qmin (r*u) (s*v)) (Qmin (r*v) (s*u)))
    (Qmax (Qmax (r*u) (s*v)) (Qmax (r*v) (s*u)))
    (QIproduct_nonempty I J) : QI.
Definition QIzero := QImake 0 0 (Qle_refl 0) : QI.
Definition QImagnitude (I:QI) := QImax I (QIminus I).
Definition QIquotient1 (J:QI) (J_correct:not (QIzero ~ (QImagnitude J))) :=
```

```
  QImake (Qinv (QIright J))(Qinv (QIleft J))(QIquotient_nonempty _ J_correct).
Definition QIquotient (I J:QI)(J_correct:not (QIzero ~ (QImagnitude J))) :=
  QIproduct I (QIquotient1 J J_correct).
```

We have proved the following properties:

```
Lemma QIsum_assoc : forall I J K:QI, ((I+J)+K) == (I+(J+K)).
Lemma QIsum_comm : forall I J:QI, (I+J) == (J+I).
Lemma QIeq_c : forall I J:QI, I==J -> I~J.
Lemma QIsum_assoc_c : forall I J K:QI, ((I+J)+K) ~ (I+(J+K)).
Lemma QIsum_comm_c : forall I J:QI, (I+J) ~ (J+I).
Lemma QIproduct_comm : forall I J:QI, (I*J) == (J*I).
Lemma QIproduct_comm_c : forall I J:QI, (I*J) ~ (J*I).
Lemma QIproduct_assoc : forall I J K:QI, ((I*J)*K) == (I*(J*K)).
```

The last one is actually not proved, because it requires a hand-written handling of a lot of conceptually trivial cases. There are many other properties which one might state and prove in order to get a complete library, but we did not want to state anything that we might not use for proving the completeness theorem.

We also have the following:

```
Lemma QIel_sum : forall I J:QI, forall x y:Q,
  QIel x I -> QIel y J -> QIel (x+y) (I+J).
Lemma QIel_product : forall I J:QI, forall x y:Q,
  QIel x I -> QIel y J -> QIel (x*y) (I*J).
Theorem QIsum_c_compat : forall I J K L:QI, I~K -> J~L -> (I+J)~(K+L).
Theorem QIproduct_c_compat : forall I J K L:QI, I~K -> J~L -> (I*J)~(K*L).
Theorem QIc_distrib : forall i j k I J K L:QI, i~I -> j~J -> k~K -> k~L ->
  ((i+j)*k) ~ ((I*K)+(J*L)).
```

The proofs of ones involving the product have not been completed because they require a lot of formalistic transoformations.

2.1.4. *Length.*

```
Definition QIlength (I:QI) := (QIright I - QIleft I)%Q.
```

```
Theorem QIsum_length : forall I J:QI, (QIlength (I+J) == QIlength I + QIlength J)%Q.
Theorem QIdifference_length : forall I J:QI, (QIlength (I-J) == QIlength I + QIlength J)%Q
```

The concept of length will be necessary in what follows.

2.2. **Theory of sets of intervals.** We now generalize to *sets* of intervals. The goal of this subsection will be to present real numbers as special kinds of sets of intervals, namely sets that are fine and consistent.

Type theoretically (in the calculus of constructions), a set of intervals will be represented by the predicate:

```
Definition QIS := QI -> Prop.
```

If p is of type QIS and I of type QI, we write $p\ I$ when we want to say that the predicate p holds for the interval I, i.e. that the interval I is in the set p.

2.2.1. *Order, arithmetic, consistency and fineness. Real numbers.* Two sets will be *consistent* when any two belonging intervals are. A set will be *smaller than* another set when there exist belonging intervals, the first completelly left to the second. The *smaller-or-equal* relation on sets is a generalisation of the less-than-or-consistent relation on intervals:

```
Definition QISc (p q:QIS) := forall I J:QI, p I -> q J -> QIc I J.
Definition QISlt (p q:QIS) := exists2 I:QI, p I & exists2 J:QI, q J & QIlt I J.
Definition QISle (p q:QIS) := forall I J:QI, p I -> q J -> QIlc I J.
```

A set of intervals will be *fine* if we can find a belonging interval of arbitrarily small length:

```
Definition QISfine (p:QIS) := forall epsilon:Q, (0<epsilon)%Q ->
  exists I:QI, p I /\ (QIlength I < epsilon)%Q.
```

The arithmetic on sets of intervals is again a simple generalisation of arithmetic on intervals:

```
Definition QISsum (p q:QIS) (K:QI) :=
  exists2 I:QI, p I & exists2 J:QI, q J & K = (I + J).

Definition QISminus (p:QIS) (K:QI) := exists2 I:QI, p I & K = (-I).

Definition QISdifference (p q:QIS) (K:QI) :=
  exists2 I:QI, p I & exists2 J:QI, q J & K = (I - J).

Definition QISproduct (p q:QIS) (K:QI) :=
  exists2 I:QI, p I & exists2 J:QI, q J & K = (I * J).

Definition QISmagnitude (p:QIS) (K:QI) :=
  exists2 I:QI, p I & K = QImagnitude I.

Definition QISzero (K:QI) := K == QIzero.

Definition QISquotient1 (p:QIS) (p_gt_zero:not (QISc QISzero p))
  (K:QI) := forall I:QI,
    forall I_gt_zero:not (QIzero ~ (QImagnitude I)),
      p I -> K == (QIquotient1 I I_gt_zero).

Definition QISquotient (q p:QIS) (p_gt_zero:not (QISc QISzero p))
  (K:QI) := forall J I:QI,
    forall I_gt_zero:not (QIzero ~ (QImagnitude I)),
      q J -> p I -> K == (QIquotient J I I_gt_zero).

Definition QISmax (p q:QIS) (K:QI) :=
  exists2 I:QI, p I & exists2 J:QI, q J & K = QImax I J.

Definition QISmin (p q:QIS) (K:QI) :=
  exists2 I:QI, p I & exists2 J:QI, q J & K = QImin I J.
```

2.2.2. *Properties.*

```
Theorem QISle_not_lt : forall p q:QIS, QISle p q -> not (QISlt q p).
Theorem QISle_le_c : forall p q:QIS, QISle p q -> QISle q p -> QISc p q.
Theorem QISlt_lt_lt : forall p w q:QIS,
  QISc w w -> QISlt p w -> QISlt w q -> QISlt p q.
Theorem QISlt_le_lt : forall p w q:QIS,
```

```
  QISfine q -> QISlt p w -> QISle w q -> QISlt p q.
Lemma QISsum_fine : forall p q : QIS, fine p -> fine q -> fine (p+q).
Lemma QISsum_cons : forall p q : QIS, cons p -> cons q -> cons (p+q).
Lemma QISproduct_cons : forall p q : QIS, cons p -> cons q -> cons (p*q).
Lemma QISmax_cons : forall p q : QIS, cons p -> cons q -> cons (QISmax p q).
Lemma QISmax_fine : forall p q : QIS, fine p -> fine q -> fine (max p q).
Lemma QISmagnitude_fine : forall p : QIS, fine p -> fine (QISmagnitude p).
```

The proofs of all these are somewhat formalistic and are best left to the reader to read from the Coq proof script. The fineness of the product is not proved there, because of the more complicated definition of the product.

2.2.3. *'Real' numbers.* Now we can formally define 'real' numbers as fine and consistent sets of intervals with rational end-points:

```
Record R : Type := Rmake {
  Rset : QIS;
  Rfine : QISfine Rset;
  Rconsistent : QISc Rset Rset }.
```

The equality of 'real' numbers will arise through our consistency relation:

```
Definition Req (p q:R) := QISc (Rset p) (Rset q).
```

Here (Rset x) extracts the carrier set from x. The order and arithmetic on 'real' numbers is defined through the order and arithmetic on sets of intervals, ie. like this:

```
Notation Rlt := (fun p q : R => QISlt (Rset p) (Rset q)).
Notation Rgt := (fun p q : R => QISlt (Rset q) (Rset p)).
Notation Rc := (fun p q : R => QISc (Rset p) (Rset q)).
Notation Rlc := (fun p q : R => QISle (Rset p) (Rset q)).
Notation Rle := (fun p q : R => (Rlt p q) \/ (Req p q)).
...
```

At this point we can use the properties from the previous subsection, as well as properties which should have been there (but are not due to time constraints), to get that the arithmetic is well-defined (sum of real numbers is a real number, ...), that the order is really a total order, and that Rmin and Rmax create a lattice. This point, in case the formalisation was finished, would show that we have an Archimedean ordered field and a lattice.

2.3. **Completeness and limits.** To show that 'real' numbers are really real numbers we need to show their completeness. This is acheaved through a generalisation of the concept of Cauchy sequence: the *number-like sets* of intervals with real end-points.

2.3.1. *Generalising the theories of intervals and sets of intervals.* Analogously to the previous two subsections one can define the theories of intervals with real end-points, and of sets of intervals with real end-points.

This is in practice (in Coq) not easy to do. The main reason is that we need to have a library for our 'real' numbers that would mirror Coq's rational number library. An additional difficulty is the fact that the later library itself is not sufficiently rich.

For now, in order to achieve the "greater" goal of proving formally the completeness, we just copy and paste the definitions and theorems about intervals with rational end-points into ones with real end-points. This development can be seen in appendices A.4 – A.5.

Another, better, development would be to abstract from the concrete rational/real-endpoints into endpoints over a Set. An attempt at this can be seen in appendix A.8. This approach was not followed for two reasons. First, the additional layer of abstraction makes things beurocratically more difficult to prove. Second, trying this approached revealed a bug in the Coq kernel, related to handling of universes, and thus the continuation of the approach had to wait for the bug to be fixed.[1]

2.3.2. *The completeness theorem.* Counting on the reader having browsed the mentioned appendices, we define a number-like set of intervals with 'real' endpoints:

```
Record NL : Type := NLmake {
  NLset : RIS;
  NLfine : RISfine NLset;
  NLconsistent : RISc NLset NLset }.
```

The limit functor gives, for each number-like set Λ, an QIS such that an interval I is inside the QIS if there is a interval K (with 'real' end-points) which is inside both Λ and I:

```
Definition Lim (Lambda:NL)(I:QI) :=
  exists K:RI, (NLset Lambda) K /\ RIinQI K I.
```

It should be obvious that each Cauchy sequence of 'real' numbers is a number-like set. The completeness theorem is now:

```
Theorem completeness : forall Lambda:NL,
  QISfine (Lim Lambda) /\ QISc (Lim Lambda) (Lim Lambda).
```

Both the fineness and consistency of `Lim Lambda` follow directly from the fineness and consistency of 'real' numbers. The proof has, however not been formalised, due to the lack of library for 'real' numbers.

3. Defining Euler's number

We define the number e by defining e^{-1}, as is usually done, for computational purposes, as the series $\sum_{l=0}^{\infty}(-1)^l/l!$. The l-finite partial sum is defined as

```
Definition Eapprox (l:positive) :=
 let f := fun arg =>
  let sum := snd arg in
  let n := fst (fst arg) in
  let nfact := snd (fst arg) in
  let sum' := Qred (sum + if (Peven n) then (-(1#nfact)) else (1#nfact))
  in (Psucc n, Pmult n nfact, sum')
 in iter_pos l (positive*positive*Q) f (2%positive, 1%positive, (1#1)).
```

We just note that Qred normalises the partial sum by factoring with the GCD of the denominator and numerator. This has significant impact on the efficiency of computations, because as the representation of the sum in Coq increases manipulating it becomes more expensive.

[1]This is done in the latest version of Coq available throught the SVN repository.

Now we define the set of intervals which will become, in our framework, the real number e, once we prove its fineness and consistency. (this is not the only possible way to define E, but it seems to be the simplest one to prove properties about)

```
Definition E : QIS :=
  fun I:QI =>
    exists n:positive, I = (Einterval n).
```

where

```
Definition Einterval' (n:positive) : Q*Q :=
  let c := (Eapprox (2*n-1)) in
    (snd c, ((snd c)+(1#(snd (fst c))))%Q).
Theorem Einterval'_nonempty : forall n:positive,
  (fst (Einterval' n) <= snd (Einterval' n))%Q.
Definition Einterval (n:positive) : QI :=
  let bounds:=Einterval' n in
    QImake (fst bounds) (snd bounds) (Einterval'_nonempty n).
```

3.1. **Consistency of** E.

```
Theorem Econsistent : QISc E E.
```

Proof. We have to prove that for any two intervals I and J which belong to e, $I \sim J$ (I and J are consistent, ie. intersect). Formally, what we have to prove is

```
Qmax (QIleft I) (QIleft J) <= Qmin (QIright I) (QIright J)
```

The proofs of $I \in e$ and $J \in e$ give us k and n such that $I = Einterval\ n$ and $J = Einterval\ k$. We make a case distinction:

$k = n$ Then $I = J$ and we have to prove

```
Qmax (QIleft J) (QIleft J) <= Qmin (QIright J) (QIright J)
```

which we easily obtain from the proof that left is left of right, a constituent of the definition of any interval.

$k < n$ Using lemmas Enested-left and Enested-right bellow, we obtain that $QIleftI < QIleftJ$ and $QIrightJ < QIrightI$. From max being an upper bound of its operands and min their lower bound we get the required.

$k > n$ The same proof as in the previous case.

□

The lemmas necessary for the previous prove express the fact that E is a set of nested intervals.

```
Lemma Enested_left_succ : forall n:positive,
  (QIleft (Einterval n) < QIleft (Einterval (Psucc n)))%Q.
Lemma Enested_left : forall k n:positive, Pgt k n ->
  (QIleft (Einterval n) < QIleft (Einterval k))%Q.
Lemma Enested_right_succ : forall n:positive,
  QIright (Einterval (Psucc n)) < QIright (Einterval n).
Lemma Enested_right : forall k n:positive, Pgt k n ->
  (QIright (Einterval k) < QIright (Einterval n))%Q.
```

Proof of `Enested-left`*.* By induction on k. The case $k = 1$ is impossible (we work with the type `positive`). For the induction case, given the two hypotheses

```
forall n : positive, Pgt p n ->
  QIleft (Einterval n) < QIleft (Einterval p)
```

and

```
Pgt (Psucc p) n
```

we should prove that

```
QIleft (Einterval n) < QIleft (Einterval (Psucc p))
```

We prove it by proving

```
QIleft (Einterval n) <= QIleft (Einterval p)
```

and

```
QIleft (Einterval p) < QIleft (Einterval (Psucc p))
```

The second goal is more complex and left for the separate lemma `Einterval-left-succ` bellow. The first is handled by using the following case distinction coming from the second hypothesis:

$p > n$ Use the first hypothesis.
$p = n$ Use the equality.

Before ending the proof we want to remark that in order to make the distinction, and many such other conceptually trivial inferences, we had to write a couple of lines of obfuscated code, as in the Coq standard library there is very limited support for the type `positive`. □

Proof of `Enested-left-succ`. After unfolding the definitions one has to prove that

```
forall n : positive,
   (snd (Eapprox (2 * n - 1)) < snd (Eapprox (2 * (n + 1) - 1)))%Q
```

Eapprox is defined in terms of the iterator `iter-pos` from the standard library. If we denote by f the function that is iterated over, and by x the sum which is the left hand side of the above inequation, and use lemma `iter-pos-plus` from the standard library, we have to prove that the $x < ffx$. This is done in lemma `odd-F-twice` which in turn requires that $2n-1$ be odd. This last thing is of course true, but to prove that in Coq is a bit technical and not adequate for explanation. The same is true for the proof of `odd-F-twice`; the reader should run the proofs in Coq if he is interested. □

The proofs of `Enested-right` and `Enested-right-succ` are analogous to the last two.

During all these proofs there are various simple equations and inequations to be solved. For that we use the special Coq tactics `ring` and `omega` which implement the decidability algorithm for Pressburger arithmetic. As this is only implemented for the types *nat* and Z, we have to inject into those types by using the following of our lemmas: `Zpos-Peq-inject`, `Zpos-Pgt-inject` and `Zpos-minus-distr`; and also the lemmas `Zpos-*` from the standard library.

3.2. **Fineness of** `E`**.** The goal is to prove the following:

```
Theorem Efine : QISfine E.
```

ie

```
Lemma Efine' :
  forall epsilon : Q, (0 < epsilon)%Q ->
    exists k:positive,
      E (Einterval k) /\ (QIlength (Einterval k) < epsilon)%Q.
```

Thus, we have to find, for each ϵ an interval in e with a length smaller than ϵ. When writing this in a programming language, one would normally calculate the finite partial sum of e until the required precision is achieved and then stop the computation. Although it is clear that this process terminates, and one can only write terminating programs in Coq, the author found out that it is not so easy to write it efficiently in Coq, as one needs to formally show that the computation is well founded.

As calculating which term of the partial sum achieves the required precision is not computationally expensive in comparison to computing the partial sum itself, and as this calculation is easy to write down with structural recursion, we instead first do this and then compute the partial sum to the desired precision.

We write this in Coq like this:

```
Definition estimate (eps:Q) : positive*positive := estimate' (Qden eps) 1 1 eps.
```

where

```
Fixpoint estimate' (n k kfact:positive) (eps:Q) {struct n} : positive*positive :=
 if (Qlt_bool (1#kfact) eps)
  then (k,kfact)
  else
   let k':=k+1 in
   let kfact':=k*kfact in
   match n with
    | xH => (k,kfact)
    | xO n' => estimate' n' k' kfact' eps
    | xI n' => estimate' n' k' kfact' eps
   end.
```

This definition may seem a bit ad hoc, but it also seems to work. `estimate'` decreases structurally on the denominator of the precision ϵ, calculating the factorial until the reciprocal value of the factorial becomes less than epsilon. In computational test estimate always gave a sufficient estimate for the required ϵ precision. We are convinced that it works even in cases not tested because for positive values the factorial increases faster than the exponential function and the structural recursion decreases at most exponentially.

What we have to prove now is that (`estimate` ϵ)-th Einterval's length is smaller than ϵ:

```
Lemma Einterval_length : forall (eps:Q), (0 < eps)%Q ->
  (QIlength (Einterval (estimate eps)) < eps)%Q.
```

or, with definitions unfolded:

```
Lemma Einterval_length' : forall (eps:Q), (0 < eps)%Q ->
  (1 # snd (fst (Eapprox (fst (estimate eps)))) < eps)%Q.
```

ϵ	vm-compute in (estimate ϵ)	vm-compute in (Eapprox (estimate ϵ))
$1/10^{12}$	0.008001	0.020001
$1/10^{24}$	0.008001	0.104006
$1/10^{90}$	0.048003	3.68823
$1/10^{180}$	0.136009	26.481655

TABLE 1. Performance of 'Eappox' and 'estimate' in seconds

Remark now that `snd (fst (Eapprox N))` computes the factorial of N. The same is true for `snd (estimate eps)`. Therefore, we could finish the proof with the following two lemmas:

```
Lemma lemma1 : forall (eps:Q), (0 < eps)%Q ->
  (snd (estim eps)) = (snd (fst (Eapprox ((fst (estim eps)))))).
Lemma lemma2 : forall (eps:Q), (0 < eps)%Q ->
  ((1#(snd (estimate eps))) < eps)%Q.
```

Now, both of these lemmas do not have (formal) proofs. Lemma 2 should be possible to be finished. If one unfolds the definition of estimate', one sees that the function returns a value either in the if-then case when the goal of lemma is met, or in the if-else case when the counter n has reached 1. Possibly, for this second case, one might have to prove that 1 is never reached (except maybe in trivial cases).

As for lemma 1, one should prove that both sides of the equation compute the factorial function. As far as we can say at this moment, this should be done by induction on the denominator of ϵ together with the following lemma:

```
Lemma lem1 : forall (p:positive)(z:Z), (z#p)<1 ->
  (fst (estimate' p 1 1 (z#p))) =
  (fst (estimate' (Psucc p) 1 1 (z#(Psucc p)))).
```

The proof of this remains unfinished. Thus, the 'real number' `E` is not yet the real number e.

3.3. **Speed of computation.** A goal of this experiment has been to produce a fast way to compute Euler's number. This had effect on the choice of types to work in (`positive` instead of `N`) and the complexity of algorithms.

We give a table of data of benchmarks carried out on our own 800 MHz Pentium III running the development version of Coq 8.1 on Debian Linux 4.0.

Appendix A. Actual definitions and lemmas

A.1. **Intervals with rational endpoints.** Module QI

Implementing reals la Stolzenberg

QI.v – Rational intervals

Danko Ilik, svn revision: $*Id*: *QI.v* 81 2007-09-09 13:54:12*Z* *danko* $

Require Import *QArith.*
Open Scope Q_scope.

Some additional constructions over Q not in the standard library
Section *Q_extras.*
Definition *Qlt_bool* (*x y*:*Q*) :=
match (*x* ?= *y*)%*Q* with

— $Lt \Rightarrow true$
— _ $\Rightarrow false$
end.

Definition *Qle_bool* $(x\ y{:}Q)$:=
match $(x\ ?{=}\ y)\%Q$ with
— $Lt \Rightarrow true$
— $Eq \Rightarrow true$
— _ $\Rightarrow false$
end.

Definition *Qmax* $(x\ y{:}Q)$:= if (*Qlt_bool* x y) then y else x.
Definition *Qmin* $(x\ y{:}Q)$:= if (*Qlt_bool* x y) then x else y.
Lemma *Qlt_bool_refl* : $\forall$ $x{:}Q$, *Qlt_bool* x x = *false*.

Lemma *Qmax_case* : $\forall$ $(n\ m{:}Q)$ $(P{:}Q \to$ Type$)$, $P\ n \to P\ m \to P$ (*Qmax* n m).

Lemma *Qmin_case* : $\forall$ $(n\ m{:}Q)$ $(P{:}Q \to$ Type$)$, $P\ n \to P\ m \to P$ (*Qmin* n m).

Lemma *Qmax_lub* : $\forall$ $n\ m\ p{:}Q$, $n \le p \to m \le p \to$ *Qmax* n $m \le p$.

Lemma *Qmin_glb* : $\forall$ $n\ m\ p{:}Q$, $p \le n \to p \le m \to p \le$ *Qmin* n m.

Lemma *Qlt_Qlt_bool*: $\forall$ $p\ q{:}Q$, $p<q \to$ *Qlt_bool* p q = *true*.

Lemma *Qmax_determined* : $\forall$ $p\ q{:}Q$, $p<q \to$ *Qmax* p $q = q$.

Lemma *Qmin_determined* : $\forall$ $p\ q{:}Q$, $p<q \to$ *Qmin* p $q = p$.

Require Import *Bool*.

Lemma *Qmin_sym* : $\forall$ $p\ q{:}Q$, *Qmin* p q = *Qmin* q p.

Lemma *Qmax_sym* : $\forall$ $p\ q{:}Q$, *Qmax* p q = *Qmax* q p.

Lemma *Qmax_ub* : $\forall$ $p\ q{:}Q$, $p \le$ *Qmax* p q.

Lemma *Qmax_ub'* : $\forall$ $p\ q{:}Q$, $q \le$ *Qmax* p q.

Lemma *Qmin_lb* : $\forall$ $p\ q{:}Q$, *Qmin* p $q \le p$.

Lemma *Qmin_lb'* : $\forall$ $p\ q{:}Q$, *Qmin* p $q \le q$.

Definition *Qabs* $(q{:}Q)$:= *Qmake* (*Zabs* (*Qnum* q)) (*Qden* q).
End *Q_extras*.
Hint *Resolve Qmax_ub*.
Hint *Resolve Qmax_ub'*.
Hint *Resolve Qmin_lb*.
Hint *Resolve Qmin_lb'*.

The theory of rational intervals
Record *QI* : Set :=
QImake { *QIleft* : Q;
QIright : Q;
QInonempty : (*QIleft*$\le$*QIright*)$\%Q$ }.

Delimit Scope QI_scope with *QI*.
Open Scope QI_scope.

Definition *QIlt* $(I\ J{:}QI)$:= (*QIright* I) < (*QIleft* J).
Notation *QIgt* := (fun I J : $QI \Rightarrow$ *QIlt* J I).

Definition $QIel\ (x{:}Q)(I{:}QI) := ((QIleft\ I)¡=x) \wedge (x¡=(QIright\ I))$.
Definition $QIsubset\ (I\ J{:}QI) := \forall\ x{:}Q,\ QIel\ x\ I \rightarrow QIel\ x\ J$.
Definition $QIeq\ (I\ J{:}QI) := (QIleft\ I == QIleft\ J) \wedge (QIright\ I == QIright\ J)$.
Definition $QIc\ (I\ J{:}QI) :=$ (** *consistency* *)
$(Qmax\ (QIleft\ I)\ (QIleft\ J)) \leq (Qmin\ (QIright\ I)\ (QIright\ J))$.
Infix "==" := $QIeq$ (*at level* 70, *no associativity*) : *QI_scope*.
Infix "<" := $QIlt$: *QI_scope*.
Infix "¿" := $QIgt$: *QI_scope*.
Infix "∼" := QIc (*at level* 70) : *QI_scope*.

Definition *QIlt_bool* $(I\ J{:}QI) :=$
match $((QIright\ I)\ ?= (QIleft\ J))\%Q$ with
— $Lt \Rightarrow true$
— _ $\Rightarrow false$
end.
Section *Q_intervals_order_and_consistency*.

Definition $QIsingleton\ (x{:}Q) := QImake\ x\ x\ (Qle\_refl\ x) : QI$.

Lemma *QIlt_trichotomy'* : $\forall\ I\ J{:}QI,\ \{I<J\} + \{I\sim J\} + \{J<I\}$.

Lemma *QIlt_trichotomy* : $\forall\ I\ J{:}QI,\ (I<J) \vee (I\sim J) \vee (J<I)$.
Hint *Resolve QIlt_trichotomy.*

Definition $QIlc\ (I\ J{:}QI) := I<J \vee I\sim J$.
Infix "<∼" := $QIlc$ (*at level* 70) : *QI_scope*.

Lemma *QIlc_not_gt* : $\forall\ I\ J{:}QI,\ (I<\sim J) \leftrightarrow not\ (I¿J)$.

Lemma *QIc_sym* : $\forall\ I\ J{:}QI,\ I\sim J \rightarrow J\sim I$.
Hint *Resolve QIc_sym.*

Lemma *QIc_lc* : $\forall\ I\ J{:}QI,\ I\sim J \leftrightarrow I<\sim J \wedge J<\sim I$.

Lemma *QIlt_trans* : $\forall\ I\ J\ K{:}QI,\ I<J \rightarrow J<K \rightarrow I<K$.
Hint *Resolve QIlt_trans.*

Lemma *QIlt_lc_lt* : $\forall\ I\ J\ K\ L{:}QI,\ I<J \rightarrow J<\sim K \rightarrow K<L \rightarrow I<L$.

Lemma *QImax_nonempty* : $\forall\ I\ J{:}QI$,
$(Qmax\ (QIleft\ I)\ (QIleft\ J)) \leq (Qmax\ (QIright\ I)\ (QIright\ J))$.

Lemma *QImin_nonempty* : $\forall\ I\ J{:}QI$,
$(Qmin\ (QIleft\ I)\ (QIleft\ J)) \leq (Qmin\ (QIright\ I)\ (QIright\ J))$.

Definition $QImax\ (I\ J{:}QI) := QImake$
$(Qmax\ (QIleft\ I)\ (QIleft\ J))\ (Qmax\ (QIright\ I)\ (QIright\ J))$
$(QImax\_nonempty\ I\ J) : QI$.

Definition $QImin\ (I\ J{:}QI) := QImake$
$(Qmin\ (QIleft\ I)\ (QIleft\ J))\ (Qmin\ (QIright\ I)\ (QIright\ J))$
$(QImin\_nonempty\ I\ J) : QI$.

Lemma *QImax_el* : $\forall\ I\ J{:}QI,\ \forall\ x\ y{:}Q$,
$QIel\ x\ I \rightarrow QIel\ y\ J \rightarrow QIel\ (Qmax\ x\ y)\ (QImax\ I\ J)$.

Lemma *QImin_el* : $\forall\ I\ J{:}QI,\ \forall\ x\ y{:}Q$,
$QIel\ x\ I \rightarrow QIel\ y\ J \rightarrow QIel\ (Qmin\ x\ y)\ (QImin\ I\ J)$.

Lemma $QImax\_c$: $\forall$ I J K L:QI, $I\sim J \rightarrow K\sim L \rightarrow QImax\ I\ K == QImax\ J\ L$.

Lemma $QImax\_lt$: $\forall$ I J K:QI, $(QImax\ I\ J)<K \leftrightarrow I<K \wedge J<K$.

Lemma $Qmax\_one$: $\forall$ p q:Q, $Qmax\ p\ q = p \vee Qmax\ p\ q = q$.
Lemma $QIlt\_max$: $\forall$ I J K:QI, $K<(QImax\ I\ J) \leftrightarrow K<I \vee K<J$.

Definition $QIle$ $(I\ J{:}QI) := I<J \vee I{=}{=}J$.
Infix "$\leq$" := $QIle$ (*at level* 70) : QI_scope.
Lemma $QIle\_max$: $\forall$ I J:QI, $I \leq (QImax\ I\ J)$.

Definition $QIbetween$ $(K\ I\ J{:}QI) := QImin\ I\ J <\sim K \wedge K <\sim QImax\ I\ J$.

Lemma $QIbetween\_lc$: $\forall$ K I J:QI,
  $QIbetween\ K\ I\ J \leftrightarrow (I<\sim K \wedge K<\sim J) \vee (J<\sim K \wedge K<\sim I)$.

Lemma $QIbetween\_transitivity$: $\forall$ K I J L:QI,
  $QIbetween\ K\ I\ J \rightarrow QIbetween\ L\ K\ J \rightarrow QIbetween\ L\ I\ J$.

Definition $QI\_sequence := nat \rightarrow QI$.

Open Scope nat_scope.
Lemma $QIbetween\_countable\_transitivity$: $\forall$ I:$QI\_sequence$,
  ($\forall$ n:nat, $QIbetween\ (I(n+2))\ (I(n))\ (I(n+1))) \rightarrow$
  $\forall$ n m:nat, (m¿n) $\rightarrow QIbetween\ (I(m))\ (I(n))\ (I(n+1))$.
Close Scope nat_scope.

End $Q\_intervals\_order\_and\_consistency$.

Definition $QIsum$ $(I\ J{:}QI) :=$
  $QImake\ (QIleft\ I + QIleft\ J)\ (QIright\ I + QIright\ J)$
  $(Qplus\_le\_compat$ _ _ _ _ $(QInonempty\ I)\ (QInonempty\ J))$: QI.

Definition $QIminus$ $(I{:}QI) := QImake$ (- $QIright\ I$) (- $QIleft\ I$)
  $(Qopp\_le\_compat$ _ _ $(QInonempty\ I))$: QI.

Definition $QIdifference$ $(I\ J{:}QI) := QIsum\ I\ (QIminus\ J)$: QI.

Lemma $Qmin\_lt\_max$: $\forall$ p q:Q, $Qmin\ p\ q \leq Qmax\ p\ q$.

Lemma $QIproduct\_nonempty$: $\forall$ I J:QI,
  let $r{:=}QIleft\ I$ in
    let $s{:=}QIright\ I$ in
      let $u{:=}QIleft\ J$ in
        let $v{:=}QIright\ J$ in
          $(Qmin\ (Qmin\ (r\times u)\ (s\times v))\ (Qmin\ (r\times v)\ (s\times u))) \leq$
          $(Qmax\ (Qmax\ (r\times u)\ (s\times v))\ (Qmax\ (r\times v)\ (s\times u)))$.

Definition $QIproduct$ $(I\ J{:}QI) :=$
  let $r{:=}QIleft\ I$ in
    let $s{:=}QIright\ I$ in
      let $u{:=}QIleft\ J$ in
        let $v{:=}QIright\ J$ in
          $QImake$
          $(Qmin\ (Qmin\ (r\times u)\ (s\times v))\ (Qmin\ (r\times v)\ (s\times u)))$
          $(Qmax\ (Qmax\ (r\times u)\ (s\times v))\ (Qmax\ (r\times v)\ (s\times u)))$
          $(QIproduct\_nonempty\ I\ J)$
          : QI.

Definition *QIzero* := *QImake* 0 0 (*Qle_refl* 0) : *QI*.

Definition *QImagnitude* (*I*:*QI*) := *QImax* *I* (*QIminus* *I*).
Lemma *QIquotient_nonempty* :
∀ *J*:*QI*, ∀ *J_correct*:(*not* (*QIzero* ∼ (*QImagnitude* *J*))),
((*Qinv* (*QIright* *J*)) ≤ (*Qinv* (*QIleft* *J*)))%*Q*.

Definition *QIquotient1* (*J*:*QI*) (*J_correct*:*not* (*QIzero* ∼ (*QImagnitude* *J*))) :=
QImake (*Qinv* (*QIright* *J*))(*Qinv* (*QIleft* *J*))(*QIquotient_nonempty* _ *J_correct*).

Definition *QIquotient* (*I* *J*:*QI*)(*J_correct*:*not* (*QIzero* ∼ (*QImagnitude* *J*))) :=
QIproduct *I* (*QIquotient1* *J* *J_correct*).

Infix "+" := *QIsum* : *QI_scope*.
Notation "- *x*" := (*QIminus* *x*) : *QI_scope*.
Infix "-" := *QIdifference* : *QI_scope*.
Infix "×" := *QIproduct* : *QI_scope*.
Notation "/ *x*" := (*QIquotient1* *x*) : *QI_scope*.
Infix "/" := *QIquotient* : *QI_scope*.
Section *Q_intervals_arithmetic_properties*.

Lemma *QI_minus_minus* : ∀ *J*:*QI*, - - *J* = *J*.

Lemma *QIquotient1_correct* : ∀ *J*:*QI*, ∀ *J_correct*:*not* (*QIzero* ∼ (*QImagnitude* *J*)),
not (*QIzero* ∼ (*QImagnitude* (*QIquotient1* *J* *J_correct*))).

Lemma *QIquotient1_quotient1*: ∀ *J*:*QI*, ∀ *J_correct*:*not* (*QIzero* ∼ (*QImagnitude* *J*)),
let *qJ_correct* := *QIquotient1_correct* *J* *J_correct*
in (*QIquotient1* (*QIquotient1* *J* *J_correct*) *qJ_correct*) == *J*.

Lemma *QIsum_assoc* : ∀ *I* *J* *K*:*QI*, ((*I*+*J*)+*K*) == (*I*+(*J*+*K*)).
Hint *Resolve* *QIsum_assoc*.

Lemma *QIsum_comm* : ∀ *I* *J*:*QI*, (*I*+*J*) == (*J*+*I*).
Hint *Resolve* *QIsum_comm*.

Lemma *QIeq_c* : ∀ *I* *J*:*QI*, *I*==*J* → *I*∼*J*.
Hint *Resolve* *QIeq_c*.

Lemma *QIsum_assoc_c* : ∀ *I* *J* *K*:*QI*, ((*I*+*J*)+*K*) ∼ (*I*+(*J*+*K*)).
Hint *Resolve* *QIsum_assoc_c*.

Lemma *QIsum_comm_c* : ∀ *I* *J*:*QI*, (*I*+*J*) ∼ (*J*+*I*).
Hint *Resolve* *QIsum_comm_c*.

Lemma *Qmin_eq* : ∀ *p* *q* *r* *s*:*Q*, (*p*==*r*)%*Q* → (*q*==*s*)%*Q* → (*Qmin* *p* *q* == *Qmin* *r* *s*)%*Q*.
Lemma *Qmax_eq* : ∀ *p* *q* *r* *s*:*Q*, (*p*==*r*)%*Q* → (*q*==*s*)%*Q* → (*Qmax* *p* *q* == *Qmax* *r* *s*)%*Q*.

Lemma *QIproduct_comm* : ∀ *I* *J*:*QI*, (*I*×*J*) == (*J*×*I*).
Hint *Resolve* *QIproduct_comm*.

Lemma *QIproduct_comm_c* : ∀ *I* *J*:*QI*, (*I*×*J*) ∼ (*J*×*I*).

Lemma *QIproduct_assoc* : ∀ *I* *J* *K*:*QI*, ((*I*×*J*)**K*) == (*I**(*J*×*K*)).
Hint *Resolve* *QIproduct_assoc*.

Lemma $QIproduct\_assoc\_c$: $\forall$ I J K:QI, $((I{\times}J){*}K) \sim (I{*}(J{\times}K))$.

Lemma $QIel\_sum$: $\forall$ I J:QI, $\forall$ x y:Q, $QIel$ x I $\rightarrow$ $QIel$ y J $\rightarrow$ $QIel$ $(x{+}y)$ $(I{+}J)$.

Lemma $QIel\_difference$: $\forall$ I J:QI, $\forall$ x y:Q,
$QIel$ x I $\rightarrow$ $QIel$ y J $\rightarrow$ $QIel$ $(x\text{-}y)$ $(I\text{-}J)$.

Lemma $QIel\_product$: $\forall$ I J:QI, $\forall$ x y:Q,
$QIel$ x I $\rightarrow$ $QIel$ y J $\rightarrow$ $QIel$ $(x{\times}y)$ $(I{\times}J)$.

Lemma $QIel\_magnitude$: $\forall$ I:QI, $\forall$ x:Q,
$QIel$ x I $\rightarrow$ $QIel$ $(Qabs$ $x)$ $(QImagnitude$ $I)$.

Lemma $QIel\_quotient$: $\forall$ I J:QI, $\forall$ x y:Q, $QIel$ x I $\rightarrow$ $QIel$ y J $\rightarrow$
$\forall$ $J\_correct$:not $(QIzero \sim (QImagnitude$ $J))$,
$QIel$ (x/y) $((I/J)$ $J\_correct)$.

Theorem $QIsum\_c\_compat$: $\forall$ I J K L:QI, $I{\sim}K \rightarrow J{\sim}L \rightarrow (I{+}J){\sim}(K{+}L)$.

Theorem $QIproduct\_c\_compat$: $\forall$ I J K L:QI, $I{\sim}K \rightarrow J{\sim}L \rightarrow (I{\times}J){\sim}(K{\times}L)$.

Theorem $QImagnitude\_c\_compat$: $\forall$ I K:QI, $I{\sim}K \rightarrow (QImagnitude$ $I){\sim}(QImagnitude$ $K)$.

Theorem $QIc\_distrib$: $\forall$ i j k I J K L:QI, $i{\sim}I \rightarrow j{\sim}J \rightarrow k{\sim}K \rightarrow k{\sim}L \rightarrow$
$((i{+}j){*}k) \sim ((I{\times}K){+}(J{\times}L))$.

Theorem QIc_minus_zero : $\forall$ I J:QI, $I{\sim}J \leftrightarrow (I\text{-}J){\sim}QIzero$.

Definition $QIlength$ $(I{:}QI)$:= $(QIright$ I - $QIleft$ $I)\%Q$.

Theorem $QIsum\_length$: $\forall$ I J:QI, $(QIlength$ $(I{+}J)$ == $QIlength$ I + $QIlength$ $J)\%Q$.

Theorem $QIdifference\_length$: $\forall$ I J:QI, $(QIlength$ $(I\text{-}J)$ == $QIlength$ I + $QIlength$ $J)\%Q$.

Theorem $QIquotent1\_length$: $\forall$ J:QI,
$\forall$ $J\_correct$:not $(QIzero \sim (QImagnitude$ $J))$,
$\forall$ c:Q, $(0{<}c)\%Q \rightarrow (c{\leq}QIleft$ $J)\%Q \rightarrow$
$(QIlength$ $(QIquotient1$ J $J\_correct) \leq (QIlength$ $J)/(c{\times}c))\%Q$.

Theorem $QImax\_length$: $\forall$ I J:QI,
$(QIlength$ $(QImax$ I $J)$ == $Qmax$ $(QIlength$ $I)$ $(QIlength$ $J))\%Q$.

Theorem $QImagnitude\_length$: $\forall$ I:QI,
$(QIlength$ $(QImagnitude$ $I) \leq QIlength$ $I)\%Q$.

Lemma $QIlength\_monotone$ $(I$ $J{:}QI)$: $(QIle$ I $J)\%QI \rightarrow (QIlength$ $I \leq QIlength$ $J)\%Q$.

End $Q\_intervals\_arithmetic\_properties$.

A.2. **Real numbers.** Module R
Require Import *QArith.*
Require Import *QI.*
Open Scope Q_scope.
Open Scope QI_scope.

Definition *QIS* := *QI* → Prop.
Definition *QISc* (*p q*:*QIS*) := ∀ *I J*:*QI*, *p I* → *q J* → *QIc I J*.

Definition *QISlt* (*p q*:*QIS*) := *exists2 I*:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *QIlt I J*.

Definition *QISle* (*p q*:*QIS*) := ∀ *I J*:*QI*, *p I* → *q J* → *QIlc I J*.

Definition *QISsum* (*p q*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *K* = (*I* + *J*).
Definition *QISminus* (*p*:*QIS*) (*K*:*QI*) := *exists2 I*:*QI*, *p I* & *K* = (-*I*).

Definition *QISdifference* (*p q*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *K* = (*I* - *J*).

Definition *QISproduct* (*p q*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *K* = (*I* × *J*).

Definition *QISmagnitude* (*p*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *K* = *QImagnitude I*.

Definition *QISzero* (*K*:*QI*) := *K* == *QIzero*.

Definition *QISquotient1* (*p*:*QIS*) (*p_gt_zero*:*not* (*QISc QISzero p*))
(*K*:*QI*) := ∀ *I*:*QI*,
∀ *I_gt_zero*:*not* (*QIzero* ¬ (*QImagnitude I*)),
p I → *K* == (*QIquotient1 I I_gt_zero*).

Definition *QISquotient* (*q p*:*QIS*) (*p_gt_zero*:*not* (*QISc QISzero p*))
(*K*:*QI*) := ∀ *J I*:*QI*,
∀ *I_gt_zero*:*not* (*QIzero* ¬ (*QImagnitude I*)),
q J → *p I* → *K* == (*QIquotient J I I_gt_zero*).

Definition *QISmax* (*p q*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *K* = *QImax I J*.

Definition *QISmin* (*p q*:*QIS*) (*K*:*QI*) :=
exists2 I:*QI*, *p I* & *exists2 J*:*QI*, *q J* & *K* = *QImin I J*.

Definition *QISfine* (*p*:*QIS*) := ∀ *epsilon*:*Q*, (0¡*epsilon*)%*Q* →
∃ *I*:*QI*, *p I* ∧ (*QIlength I* ¡ *epsilon*)%*Q*.
Theorem *QISc_sym* : ∀ *p q*:*QIS*, *QISc p q* → *QISc q p*.

Theorem *QISc_lt_le* : ∀ *p q*:*QIS*, *QISc p p* → *QISc q q* → *QISlt p q* → *QISle p q*.

Theorem *QISle_not_lt* : ∀ *p q*:*QIS*, *QISle p q* → *not* (*QISlt q p*).

Theorem *QISnot_lt_le* : ∀ *p q*:*QIS*, *not* (*QISlt q p*) → *QISle p q*.

Theorem *QISle_le_c* : ∀ *p q*:*QIS*, *QISle p q* → *QISle q p* → *QISc p q*.

Theorem *QISlt_lt_lt* : ∀ *p w q*:*QIS*,
QISc w w → *QISlt p w* → *QISlt w q* → *QISlt p q*.

Theorem *QISlt_le_lt* : ∀ *p w q*:*QIS*,
QISfine q → *QISlt p w* → *QISle w q* → *QISlt p q*.

Theorem *QISle_le_le* : ∀ *p w q*:*QIS*,
QISfine w → *QISle p w* → *QISle w q* → *QISle p q*.

Theorem *QISc_c_c* : ∀ *p w q*:*QIS*,
QISfine w → *QISc p w* → *QISc w q* → *QISc p q*.

Theorem *QISlt_eps* : ∀ *p q*:*QIS*, *QISlt q p* →
exists2 eps:*QIS*, *QISlt QISzero eps* & *QISlt* (*QISsum q eps*) *p*.

Theorem *thm_3_25* : ∀ *p q*:*QIS*,
(∀ *eps*:*QIS*, *QISlt QISzero eps* → *QISle p* (*QISsum q eps*)) → *QISle p q*.

Delimit Scope QIS_scope with *QIS*.
Open Scope QIS_scope.

Infix "¡" := *QISlt* : *QIS_scope*.
Infix "¬" := *QISc* (*at level* 70) : *QIS_scope*.
Infix "≤" := *QISle* (*at level* 70) : *QIS_scope*.
Infix "+" := *QISsum* : *QIS_scope*.
Notation "- *x*" := (*QISminus x*) : *QIS_scope*.
Infix "×" := *QISproduct* : *QIS_scope*.

Theorem *QIS_distrib* : ∀ *w p q*:*QIS*, (*w* × (*p* + *q*)) ¬ (*w*×*p* + *w*×*q*).

Notation *cons* := (fun *p* : *QIS* ⇒ *QISc p p*).
Notation *fine* := (fun *p* : *QIS* ⇒ *QISfine p*).
Notation *max* := (fun *p q* : *QIS* ⇒ *QISmax p q*).
Notation *min* := (fun *p q* : *QIS* ⇒ *QISmin p q*).

Notation " 2 " := (2#1) : *Q_scope*.
Open Scope Q_scope.
Lemma *Qdiv_lt_compat* : ∀ *x y z*:*Q*, 0¡*z* → *x*¡*y* → *x*/*z*¡*y*/*z*.

Lemma *Qdiv_lt_compat_zero* : ∀ *x z*:*Q*, 0¡*z* → 0¡*x* → 0¡*x*/*z*.

Lemma *Qplus_lt_compat* :
∀ *x y z t*, *x*¡*y* → *z*¡*t* → *x*+*z* ¡ *y*+*t*.

Close Scope Q_scope.

Lemma *QISsum_fine* : ∀ *p q* : *QIS*, *fine p* → *fine q* → *fine* (*p*+*q*).

Lemma *QISsum_cons* : ∀ *p q* : *QIS*, *cons p* → *cons q* → *cons* (*p*+*q*).

Lemma *QISproduct_cons* : ∀ *p q* : *QIS*, *cons p* → *cons q* → *cons* (*p*×*q*).

Lemma *QISmax_cons* : ∀ *p q* : *QIS*, *cons p* → *cons q* → *cons* (*QISmax p q*).

Open Scope Q_scope.
Theorem *Qeq_le* : ∀ *x y*:*Q*, *x*==*y* → *x*≤*y*.
Close Scope Q_scope.

Lemma *QISmax_fine* : ∀ *p q* : *QIS*, *fine p* → *fine q* → *fine* (*max p q*).

Lemma *QISmagnitude_fine* : ∀ *p* : *QIS*, *fine p* → *fine* (*QISmagnitude p*).

Definition *Q2QIS* (*q*:*Q*)(*I*:*QI*) := *I*=(*QImake q q* (*Qle_refl q*)).

Lemma *Qdense* : ∀ *p*:*QIS*, *fine p* → ∃ *q*:*Q*, *p*¡*Q2QIS q*.

Open Scope Q_scope.
Lemma *Qopp_pos_lt* : ∀ *A*:*Q*, 0¡*A* → -*A* ¡ *A*.

Definition *I_oppA_A* (*A*:*Q*)(*zlA*:0¡*A*) : *QI*.

Close Scope Q_scope.

Lemma *lemma1* : ∀ *I*:*QI*, ∀ (*A* : *Q*)(*zero_lt_A*:(0 ¡ *A*)%*Q*),
(*QIlength I* ¡ *A*)%*Q* → *QIsubset I* (*I_oppA_A A zero_lt_A*).

Theorem *QISproduct_fine* : ∀ *p q* : *QIS*, *fine p* → *fine q* → *fine* (*p*×*q*).

Record *R* : Type := *Rmake* {
Rset : *QIS*;
Rfine : *QISfine Rset*;
Rconsistent : *QISc Rset Rset* }.

Definition *Req* (*p q*:*R*) := *QISc* (*Rset p*) (*Rset q*).

Delimit Scope R_scope with *R*.
Open Scope R_scope.

Notation *QISgt* := (fun *p q* : *QIS* ⇒ *QISlt q p*).
Notation *Rlt* := (fun *p q* : *R* ⇒ *QISlt* (*Rset p*) (*Rset q*)).
Notation *Rgt* := (fun *p q* : *R* ⇒ *QISlt* (*Rset q*) (*Rset p*)).
Notation *Rc* := (fun *p q* : *R* ⇒ *QISc* (*Rset p*) (*Rset q*)).
Notation *Rlc* := (fun *p q* : *R* ⇒ *QISle* (*Rset p*) (*Rset q*)).
Notation *Rle* := (fun *p q* : *R* ⇒ (*Rlt p q*) ∨ (*Req p q*)).

Infix "==" := *Req* (*at level* 70, *no associativity*) : *R_scope*.
Infix "¡" := *Rlt* : *R_scope*.
Infix "¿" := *Rgt* : *R_scope*.
Infix "¬" := *Rc* (*at level* 70) : *R_scope*.
Infix "¡˜" := *Rlc* (*at level* 70) : *R_scope*.
Infix "≤" := *Rle* (*at level* 70) : *R_scope*.

Lemma *Rmax_consistent* : ∀ *p q* : *R*,
QISc (*QISmax* (*Rset p*) (*Rset q*)) (*QISmax* (*Rset p*) (*Rset q*)).

Lemma *Rmax_fine* : ∀ *p q* : *R*, *QISfine* (*QISmax* (*Rset p*) (*Rset q*)).

Notation *Rmax* :=
(fun *p q* : *R* ⇒
Rmake (*QISmax* (*Rset p*) (*Rset q*)) (*Rmax_fine p q*) (*Rmax_consistent p q*)).

Lemma *Rmin_consistent* : ∀ *p q* : *R*,
QISc (*QISmin* (*Rset p*) (*Rset q*)) (*QISmin* (*Rset p*) (*Rset q*)).

Lemma *Rmin_fine* : ∀ *p q* : *R*, *QISfine* (*QISmin* (*Rset p*) (*Rset q*)).

Notation *Rmin* :=
(fun *p q* : *R* ⇒
Rmake (*QISmin* (*Rset p*) (*Rset q*)) (*Rmin_fine p q*) (*Rmin_consistent p q*)).

Lemma *Rle_refl* : ∀ *x*, *x*≤*x*.

Lemma *Rsum_fine* : ∀ *p q* : *R*, *QISfine* (*QISsum* (*Rset p*) (*Rset q*)).

Lemma *Rsum_consistent* : ∀ *p q* : *R*,
QISc (*QISsum* (*Rset p*) (*Rset q*)) (*QISsum* (*Rset p*) (*Rset q*)).

Notation *Rsum* := (fun *p q* : *R* ⇒
Rmake (*QISsum* (*Rset p*) (*Rset q*)) (*Rsum_fine p q*) (*Rsum_consistent p q*)).

Infix "+" := *Rsum* : *R_scope*.

Lemma *Rplus_le_compat* :
∀ *x y z t*, *x*≤*y* → *z*≤*t* → (*x*+*z*) ≤ (*y*+*t*).

Lemma *Rminus_fine* : ∀ *p* : *R*, *QISfine* (*QISminus* (*Rset p*)).

Lemma *Rminus_consistent* : ∀ *p* : *R*,
QISc (*QISminus* (*Rset p*)) (*QISminus* (*Rset p*)).

Notation *Rminus* := (fun *p* : *R* ⇒
Rmake (*QISminus* (*Rset p*)) (*Rminus_fine p*) (*Rminus_consistent p*)).
Notation "- *x*" := (*Rminus x*) : *R_scope*.

Lemma *Ropp_le_compat* : ∀ *p q*, *p*≤*q* → -*q* ≤ -*p*.

Lemma *Rdifference_fine* : ∀ *p q* : *R*, *QISfine* (*QISdifference* (*Rset p*) (*Rset q*)).

Lemma *Rdifference_consistent* : ∀ *p q* : *R*,
QISc (*QISdifference* (*Rset p*) (*Rset q*)) (*QISdifference* (*Rset p*) (*Rset q*)).

Notation *Rdifference* := (fun *p q* : *R* ⇒
Rmake (*QISdifference* (*Rset p*) (*Rset q*)) (*Rdifference_fine p q*) (*Rdifference_consistent p q*)).
Infix "-" := *Rdifference* : *R_scope*.

Lemma *Rproduct_fine* : ∀ *p q* : *R*, *QISfine* (*QISproduct* (*Rset p*) (*Rset q*)).

Lemma *Rproduct_consistent* : ∀ *p q* : *R*,
QISc (*QISproduct* (*Rset p*) (*Rset q*)) (*QISproduct* (*Rset p*) (*Rset q*)).

Notation *Rproduct* := (fun *p q* : *R* ⇒
Rmake (*QISproduct* (*Rset p*) (*Rset q*)) (*Rproduct_fine p q*) (*Rproduct_consistent p q*)).
Infix "×" := *Rproduct* : *R_scope*.

Lemma *Rzero_fine* : *QISfine QISzero*.

Lemma *Rzero_consistent* : *QISc QISzero QISzero*.

Notation *Rzero* := (*Rmake QISzero Rzero_fine Rzero_consistent*).

Lemma *Rquotient1_fine* : ∀ *p* : *R*,
∀ *p_correct*:*not* (*QISc QISzero* (*Rset p*)),
QISfine (*QISquotient1* (*Rset p*) *p_correct*).

Lemma *Rquotient1_consistent* : ∀ *p* : *R*,
∀ *p_correct*:*not* (*QISc QISzero* (*Rset p*)),
QISc (*QISquotient1* (*Rset p*) *p_correct*) (*QISquotient1* (*Rset p*) *p_correct*).

Lemma *strip_correct* : ∀ (*p* : *R*),
not (*Rc Rzero p*) → *not* (*QISc QISzero* (*Rset p*)).

Notation *Rquotient1* := (fun (*p* : *R*)(*p_correct* : *not* (*Rc Rzero p*)) ⇒
let *pc*:=*strip_correct p p_correct*
in *Rmake* (*QISquotient1* (*Rset p*) *pc*)
(*Rquotient1_fine p pc*)
(*Rquotient1_consistent p pc*)).
Notation "/ *x*" := (*Rquotient1 x*) : *R_scope*.

Lemma *Rquotient_fine* : ∀ (*p q* : *R*)(*q_correct* : *not* (*Rc Rzero q*)), *QISfine* (*QISquotient* (*Rset p*) (*Rset q*) (*strip_correct q q_correct*)).

Lemma *Rquotient_consistent* : ∀ (*p q* : *R*)(*q_correct* : *not* (*Rc Rzero q*)),
QISc (*QISquotient* (*Rset p*) (*Rset q*) (*strip_correct q q_correct*))
(*QISquotient* (*Rset p*) (*Rset q*) (*strip_correct q q_correct*)).

Notation *Rquotient* := (fun (*p q* : *R*)(*q_correct* : *not* (*Rc Rzero q*))=¿
let *qc*:=*strip_correct q q_correct*
in *Rmake* (*QISquotient* (*Rset p*) (*Rset q*) *qc*) (*Rquotient_fine p q qc*) (*Rquotient_consistent p q qc*)).
Infix "/" := *Rquotient* : *R_scope*.

Lemma *Rmagnitude_fine* : ∀ *p* : *R*, *QISfine* (*QISmagnitude* (*Rset p*)).

Lemma *Rmagnitude_consistent* : ∀ *p* : *R*,
QISc (*QISmagnitude* (*Rset p*)) (*QISmagnitude* (*Rset p*)).

Notation *Rmagnitude* := (fun *p* : *R* ⇒
Rmake (*QISmagnitude* (*Rset p*)) (*Rmagnitude_fine p*) (*Rmagnitude_consistent p*)).

A.3. **Defining Euler's number.** Module Euler

```
Require Import NArith.
Require Import QArith.
Require Import QI.
Require Import R.

Section Euler_number_intervals.
  Open Scope Q_scope.

  Definition Peven (n:positive) : bool :=
    match n with
      — xO _ ⇒ true
      — _ ⇒ false
    end.

  Definition Eapprox (l:positive) :=
    let f :=
      fun arg ⇒
        let sum := snd arg in
          let n := fst (fst arg) in
            let nfact := snd (fst arg) in
              let sum' := Qred (sum + if (Peven n) then (-(1#nfact)) else (1#nfact))
              in (Psucc n, Pmult n nfact, sum')
        in iter_pos l (positive×positive×Q) f (2%positive, 1%positive, (1#1)).

End Euler_number_intervals.

Section Which_term_of_the_series_achieves_the_require_precision.
  Open Scope positive_scope.

  Fixpoint estimate' (n k kfact:positive) (eps:Q) {struct n} : positive×positive :=
    if (Qlt_bool (1#kfact) eps)
      then (k,kfact)
      else
```

```
        let k':=k+1 in
          let kfact':=k×kfact in
            match n with
              — xH ⇒ (k,kfact)
              — xO n' ⇒ estimate' n' k' kfact' eps
              — xI n' ⇒ estimate' n' k' kfact' eps
            end.
  Definition estimate (eps:Q) : positive×positive := estimate' (Qden eps) 1 1 eps.
  Fixpoint estim' (n k kfact:positive) (eps:Q) {struct n} : positive×positive :=
    if (Qle_bool (1#kfact) eps)
      then (k,(k+1)*k×kfact)
      else
        let k':=k+1 in
          let kfact':=k×kfact in
            match n with
              — xH ⇒ (k,(k+1)*k×kfact)
              — xO n' ⇒ estim' n' k' kfact' eps
              — xI n' ⇒ estim' n' k' kfact' eps
            end.
  Definition estim (eps:Q) : positive×positive := estim' (Qden eps) 1 1 eps.
End Which_term_of_the_series_achieves_the_require_precision.
Definition Einterval' (n:positive) : Q×Q :=
  let c := (Eapprox (2×n-1)) in
    (snd c, ((snd c)+(1#(snd (fst c))))%Q).
Lemma Qineq1 : ∀ (q:Q)(k:positive), (q ≤ q+(1#k))%Q.
Theorem Einterval'_nonempty : ∀ n:positive,
  (fst (Einterval' n) ≤ snd (Einterval' n))%Q.
Definition Einterval (n:positive) : QI :=
  let bounds:=Einterval' n in
    QImake (fst bounds) (snd bounds) (Einterval'_nonempty n).
Definition E : QIS :=
  fun I:QI ⇒
    ∃ n:positive, I = (Einterval n).
Definition Pgt (k n:positive) := ((k?=n) Eq = Gt)%positive.
Section Solving_equations_and_inequations_in_Z.
  Open Scope Z_scope.
  Lemma Zpos_Peq_inject : ∀ x y:positive, Zpos x = Zpos y → x = y.
  Lemma Zpos_Pgt_inject : ∀ x y:positive, Zpos x ¿ Zpos y → Pgt x y.
  Lemma Zpos_minus_distr : ∀ x y:positive, Pgt x y →
    Zpos (x-y) = Zpos x - Zpos y.
End Solving_equations_and_inequations_in_Z.
Open Scope positive_scope.
Definition F := (fun arg : positive × positive × Q ⇒
```

(*Psucc* (*fst* (*fst arg*)), (*fst* (*fst arg*) × *snd* (*fst arg*))%*positive*,
Qred
(*snd arg* +
(if *Peven* (*fst* (*fst arg*))
then - (1 # *snd* (*fst arg*))%*Q*
else (1 # *snd* (*fst arg*))%*Q*))%*Q*)).

Lemma *odd_F_twice* : ∀ *A*:*positive*×*positive*×*Q*,
Peven (*fst* (*fst A*)) = *false* →
(*snd A* ¡ *snd* (*F* (*F A*)))%*Q*.

Lemma *Peven_plus_two* : ∀ *k*:*positive*, *Peven k* = *Peven* (*Psucc* (*Psucc k*)).

Lemma *Enested_left_succ* : ∀ *n*:*positive*,
(*QIleft* (*Einterval n*) ¡ *QIleft* (*Einterval* (*Psucc n*)))%*Q*.

Open Scope Q_scope.

Lemma *Enested_left* : ∀ *k n*:*positive*, *Pgt k n* → (*QIleft* (*Einterval n*) ¡ *QIleft* (*Einterval k*))%*Q*.

Lemma *Einterval'_simplified* : ∀ *n*:*positive*,
snd (*Einterval' n*) == *snd* (*Eapprox* (2×*n*)).
Open Scope positive_scope.

Lemma *Podd_succ* : ∀ *k*:*positive*, *Peven k* = *true* →
Peven (*Psucc k*) = *false*.

Lemma *F_pair* : ∀ *A*:*positive*×*positive*×*Q*,
Peven (*fst* (*fst A*)) = *true* →
(*snd* (*F* (*F A*)) ¡ *snd A*)%*Q*.

Lemma *Enested_right_succ* : ∀ *n*:*positive*,
QIright (*Einterval* (*Psucc n*)) ¡ *QIright* (*Einterval n*).
Open Scope positive_scope.

Lemma *Enested_right* : ∀ *k n*:*positive*, *Pgt k n* → (*QIright* (*Einterval k*) ¡ *QIright* (*Einterval n*))%*Q*.

Theorem *Econsistent* : *QISc E E*.

Lemma *gt_quotients* : ∀ *m n*:*positive*, ((*m*?=*n*)%*positive Eq* = *Gt*) → ((1#*m*)¡(1#*n*))%*Q*.

Lemma *Einterval_in_E* : ∀ *k*:*positive*, *E* (*Einterval k*).

Lemma *lemma1* : ∀ (*eps*:*Q*), (0 ¡ *eps*)%*Q* →
(*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*)))))).

Eval compute in (*Eapprox* 1).
Lemma *lem1* : ∀ (*p*:*positive*)(*p0*:*Z*),
(*fst* (*estimate' p* 1 1 (*p0*#*p*))) = (*fst* (*estimate'* (*Psucc p*) 1 1 (*p0*#(*Psucc p*)))).

Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 1%Z 2).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 2).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 1%Z 3).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 4).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 1%Z 100).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 100).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 1%Z 100000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 100000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 1%Z 100000000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 1%Z 100000000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 2).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 2).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 3).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 3).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 4).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 4).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 100).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 100).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 100000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 100000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 3%Z 100000000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 3%Z 100000000).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 5%Z 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' p 1 1 (p0 # p))) 5%Z 1).

Eval compute in ((fun (p0:Z)(p:positive)=¿fst (estim' (Psucc p) 1 1 (p0 # Psucc p))) 5%Z 2).

Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 5).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 5).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 6).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 6).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 7).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*fst* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 7).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 100).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 100).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 100000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 100000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 1%*Z* 100000000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 1%*Z* 100000000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 4).

Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 100).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 100).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 100000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 100000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 3%*Z* 100000000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 3%*Z* 100000000).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 1).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 2).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 3).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 4).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 5).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 5).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 6).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 6).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim'* (*Psucc p*) 1 1 (*p0* # *Psucc p*))) 5%*Z* 7).
Eval compute in ((fun (*p0*:*Z*)(*p*:*positive*)=¿*snd* (*estim' p* 1 1 (*p0* # *p*))) 5%*Z* 7).

Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (1#1)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (1#2)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (1#3)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#1)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#2)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#3)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
(*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#4)).

Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
  (*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#5)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
  (*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#6)).
Eval compute in ((fun (*eps*:*Q*) ⇒ (*snd* (*estim eps*))=
  (*snd* (*fst* (*Eapprox* ((*fst* (*estim eps*))))))) (5#7)).

Lemma *lemma2* : ∀ (*eps*:*Q*), (0 ¡ *eps*)%*Q* →
  ((1#(*snd* (*estimate eps*))) ¡ *eps*)%*Q*.

Lemma *Einterval_length'* : ∀ (*eps*:*Q*), (0 ¡ *eps*)%*Q* →
  (1 # *snd* (*fst* (*Eapprox* (*fst* (*estimate eps*)))) ¡ *eps*)%*Q*.

Lemma *Einterval_length'* : ∀ (*eps*:*Q*), (0 ¡ *eps*)%*Q* →
  (1 # *snd* (*fst* (*Eapprox* (*xO* (*estimate eps*) - 1))) ¡ *eps*)%*Q*.

Lemma *Einterval_length* : ∀ (*eps*:*Q*), (0 ¡ *eps*)%*Q* →
  (*QIlength* (*Einterval* (*estimate eps*)) ¡ *eps*)%*Q*.

Lemma *Efine'* :
  ∀ *epsilon* : *Q*, (0 ¡ *epsilon*)%*Q* →
    ∃ *k*:*positive*,
      E (*Einterval k*) ∧ (*QIlength* (*Einterval k*) ¡ *epsilon*)%*Q*.

Theorem *Efine* : *QISfine E*.

A.4. **Intervals with real endpoints.** Module RI
Require Import *QI*.
Require Import *R*.
Open Scope QI_scope.
Open Scope R_scope.

Record *RI* :Type := *RImake* {
  RIleft : *R*;
  RIright : *R*;
  RInonempty : *Rle RIleft RIright* }.

Delimit Scope RI_scope with *RI*.
Open Scope RI_scope.

Definition *RIlt* (*I J*:*RI*) := (*RIright I*) ¡ (*RIleft J*).
Notation *RIgt* := (fun *I J* : *RI* ⇒ *RIlt J I*).

Definition *RIel* (*x*:*R*)(*I*:*RI*) := ((*RIleft I*)¡=*x*) ∧ (*x*¡=(*RIright I*)).
Definition *RIsubset* (*I J*:*RI*) := ∀ *x*:*R*, *RIel x I* → *RIel x J*.
Definition *RIeq* (*I J*:*RI*) := *RIsubset I J* ∧ *RIsubset J I*.
Definition *RIc* (*I J*:*RI*) := (*Rmax* (*RIleft I*) (*RIleft J*)) ≤ (*Rmin* (*RIright I*) (*RIright J*)).
Infix "==" := *RIeq* (*at level* 70, *no associativity*) : *RI_scope*.
Infix "¡" := *RIlt* : *RI_scope*.
Infix "¿" := *RIgt* : *RI_scope*.
Infix "¬" := *RIc* (*at level* 70) : *RI_scope*.

Section *R_intervals_order_and_consistency*.

Hypotheses *I J K L*:*RI*.

Definition *RIsingleton* (*x*:*R*) := *RImake x x* (*Rle_refl x*) : *RI*.

Lemma *RIlt_trichotomy'* : (*I*¡*J*) ∨ (*I*¬*J*) ∨ (*J*¡*I*).

Definition *RIlc* (*I J*:*RI*) := *I*¡*J* ∨ *I*¬*J*.
Infix "¡˜" := *RIlc* (*at level* 70) : *RI_scope*.

Lemma *RIlc_not_gt* : (*I*¡˜*J*) ↔ *not* (*I*¿*J*).

Lemma *RIc_lc* : *I*¬*J* ↔ *I*¡˜*J* ∧ *J*¡˜*I*.

Lemma *RIlt_transitivity* : *I*¡*J* ∧ *J*¡*K* → *I*¡*K*.

Lemma *RIlt_lc* : *I*¡*J* ∧ *J*¡˜*K* → *I*¡*K*.

Lemma *RIlt_lc'* : *I*¡*J* ∧ *J*¡˜*K* ∧ *K*¡*L* → *I*¡*L*.

Lemma *RImax_nonempty* : ∀ *I J*:*RI*,
(*Rmax* (*RIleft I*) (*RIleft J*)) ≤ (*Rmax* (*RIright I*) (*RIright J*)).
Lemma *RImin_nonempty* : ∀ *I J*:*RI*,
(*Rmin* (*RIleft I*) (*RIleft J*)) ≤ (*Rmin* (*RIright I*) (*RIright J*)).

Definition *RImax* (*I J*:*RI*) :=
RImake (*Rmax* (*RIleft I*) (*RIleft J*)) (*Rmax* (*RIright I*) (*RIright J*))
(*RImax_nonempty I J*) : *RI*.
Definition *RImin* (*I J*:*RI*) :=
RImake (*Rmin* (*RIleft I*) (*RIleft J*)) (*Rmin* (*RIright I*) (*RIright J*))
(*RImin_nonempty I J*) : *RI*.

Lemma *RImax_el* : ∀ *x y*:*R*,
RIel x I → *RIel y J* → *RIel* (*Rmax x y*) (*RImax I J*).
Lemma *RImin_el* : ∀ *x y*:*R*,
RIel x I → *RIel y J* → *RIel* (*Rmin x y*) (*RImin I J*).

Lemma *RImax_c* : *I*¬*J* → *K*¬*L* → *RImax I K* == *RImax J L*.

Lemma *RIc_max_min* :
(*RImax I J*)˜*J* ↔ (*RImin I J*)˜*I* ↔ *I*¡˜*J*.

Lemma *RIsubset_max_min* : *RIsubset* (*RImax I J*) *I* ∨ *RIsubset* (*RImax I J*) *J*.

Lemma *RImax_lt* : (*RImax I J*)¡*K* ↔ *I*¡*K* ∧ *J*¡*K*.

Lemma *RIlt_max* : *K*¡(*RImax I J*) ↔ *K*¡*I* ∨ *K*¡*J*.

Definition *RIle* (*I J*:*RI*) := *I*¡*J* ∨ *I*==*J*.
Infix "≤" := *RIle* (*at level* 70) : *RI_scope*.
Lemma *RIle_max* : *I* ≤ (*RImax I J*).

Definition *RIbetween* (*K I J*:*RI*) := *RImin I J* ¡˜ *K* ∧ *K* ¡˜ *RImax I J*.

Lemma *RIbetween_lc* : *RIbetween K I J* ↔ (*I*¡˜*K* ∧ *K*¡˜*J*) ∨ (*J*¡˜*K* ∧ *K*¡˜*I*).

Lemma *RIbetween_transitivity* :
RIbetween K I J → *RIbetween L K J* → *RIbetween L I J*.

Definition *RI_sequence* := *nat* → *RI*.

Open Scope nat_scope.

Lemma *RIbetween_countable_transitivity* : ∀ *I*:*RI_sequence*,
(∀ *n*:*nat*, *RIbetween* (*I*(*n*+2)) (*I*(*n*)) (*I*(*n*+1))) →
∀ *n m*:*nat*, (*m*¿*n*) → *RIbetween* (*I*(*m*)) (*I*(*n*)) (*I*(*n*+1)).
Close Scope nat_scope.

End *R_intervals_order_and_consistency.*

Definition *RIsum* (*I J*:*RI*) :=
RImake (*RIleft I* + *RIleft J*) (*RIright I* + *RIright J*)
(*Rplus_le_compat* _ _ _ _ (*RInonempty I*) (*RInonempty J*)) : *RI.*

Definition *RIminus* (*I*:*RI*) := *RImake* (- *RIright I*) (- *RIleft I*)
(*Ropp_le_compat* _ _ (*RInonempty I*)) : *RI.*

Definition *RIdifference* (*I J*:*RI*) := *RIsum I* (*RIminus J*) : *RI.*

Lemma *RIproduct_nonempty* : ∀ *I J*:*RI*,
let *r*:=*RIleft I* in
let *s*:=*RIright I* in
let *u*:=*RIleft J* in
let *v*:=*RIright J* in
(*Rmin* (*r*×*u*) (*Rmin* (*r*×*v*) (*Rmin* (*s*×*u*) (*s*×*v*)))) ≤
(*Rmax* (*r*×*u*) (*Rmax* (*r*×*v*) (*Rmax* (*s*×*u*) (*s*×*v*)))).

Definition *RIproduct* (*I J*:*RI*) :=
let *r*:=*RIleft I* in
let *s*:=*RIright I* in
let *u*:=*RIleft J* in
let *v*:=*RIright J* in
RImake
(*Rmin* (*r*×*u*) (*Rmin* (*r*×*v*) (*Rmin* (*s*×*u*) (*s*×*v*))))
(*Rmax* (*r*×*u*) (*Rmax* (*r*×*v*) (*Rmax* (*s*×*u*) (*s*×*v*))))
(*RIproduct_nonempty I J*)
: *RI.*

Definition *RIzero* := *RImake Rzero Rzero* (*Rle_refl Rzero*) : *RI.*

Definition *RImagnitude* (*I*:*RI*) := *RImax I* (*RImax* (*RIminus I*) *RIzero*).

Lemma *RIleft_correct* : ∀ *J*:*RI*,
not (*RIc RIzero* (*RImagnitude J*)) → *not* (*Rc Rzero* (*RIleft J*)).

Lemma *RIright_correct* : ∀ *J*:*RI*,
not (*RIc RIzero* (*RImagnitude J*)) → *not* (*Rc Rzero* (*RIright J*)).

Lemma *RIquotient_nonempty* :
∀ *J*:*RI*, ∀ *J_correct*:*not* (*RIc RIzero* (*RImagnitude J*)),
((*Rquotient1* (*RIright J*) (*RIright_correct J J_correct*))
≤ (*Rquotient1* (*RIleft J*) (*RIleft_correct J J_correct*)))%*R.*

Definition *RIquotient1* (*J*:*RI*) (*J_correct*:*not* (*RIc RIzero* (*RImagnitude J*))) :=
RImake
(*Rquotient1* (*RIright J*) (*RIright_correct J J_correct*))
(*Rquotient1* (*RIleft J*) (*RIleft_correct J J_correct*))
(*RIquotient_nonempty* _ *J_correct*).

Definition *RIquotient* (*I J*:*RI*)(*J_correct*:*not* (*RIc RIzero* (*RImagnitude J*))) :=

RIproduct I (RIquotient1 J J_correct).

Infix "+" := *RIsum* : *RI_scope.*
Notation "- x" := (*RIminus x*) : *RI_scope.*
Infix "-" := *RIdifference* : *RI_scope.*
Infix "×" := *RIproduct* : *RI_scope.*
Notation "/ x" := (*RIquotient1 x*) : *RI_scope.*
Infix "/" := *RIquotient* : *RI_scope.*
Section *R_intervals_arithmetic_properties.*
Hypotheses *I J K*:*RI.*

Lemma *RI_minus_minus* : - - *J* = *J.*

Lemma *RIquotient1_correct* :∀ *J_correct*:*not* (*RIzero* ¬ (*RImagnitude J*)),
not (*RIzero* ¬ (*RImagnitude* (*RIquotient1 J J_correct*))).

Lemma *RIquotient1_quotient1*:∀ *J_correct*:*not* (*RIzero* ¬ (*RImagnitude J*)),
let *qJ_correct* := *RIquotient1_correct J_correct*
in (*RIquotient1* (*RIquotient1 J J_correct*) *qJ_correct*) == *J.*

Lemma *RIsum_associative* : ((*I*+*J*)+*K*) == (*I*+(*J*+*K*)).

Lemma *RIsum_commutative* : (*I*+*J*) == (*J*+*I*).

Lemma *RIsum_associative_c* : ((*I*+*J*)+*K*) ¬ (*I*+(*J*+*K*)).

Lemma *RIsum_commutative_c* : (*I*+*J*) ¬ (*J*+*I*).

Lemma *RIproduct_associative* : ((*I*×*J*)**K*) == (*I**(*J*×*K*)).

Lemma *RIproduct_commutative* : (*I*×*J*) == (*J*×*I*).

Lemma *RIel_sum* : ∀ *x y*:*R*, *RIel x I* → *RIel y J* → *RIel* (*x*+*y*) (*I*+*J*).

Lemma *RIel_difference* : ∀ *x y*:*R*,
RIel x I → *RIel y J* → *RIel* (*x*-*y*) (*I*-*J*).

Lemma *RIel_product* : ∀ *x y*:*R*,
RIel x I → *RIel y J* → *RIel* (*x*×*y*) (*I*×*J*).

Lemma *RIel_magnitude* : ∀ *x*:*R*,
RIel x I → *RIel* (*Rmagnitude x*) (*RImagnitude I*).

Lemma *RIel_quotient* : ∀ *x y*:*R*, *RIel x I* → *RIel y J* →
∀ *y_correct*:*not* (*Rc Rzero y*),
∀ *J_correct*:*not* (*RIc RIzero* (*RImagnitude J*)),
RIel (*Rquotient x y y_correct*) (*RIquotient I J J_correct*).

Definition *RIlength* (*I*:*RI*) := (*RIright I* - *RIleft I*)%*R.*

Lemma *RIlength_monotone* (*I J*:*RI*) : (*RIle I J*)%*RI* → (*RIlength I* ≤ *RIlength J*)%*R.*

End *R_intervals_arithmetic_properties.*

A.5. **Sets of intervals with real endpoints.** Module RIS

Require Import *QI*.
Require Import *R*.
Require Import *RI*.
Open Scope R_scope.
Open Scope RI_scope.

Definition *RIS* := *RI* → Prop.
Definition *RISc* (*p q*:*RIS*) := ∀ *I J*:*RI*, *p I* → *q J* → *RIc I J*.

Definition *RISlt* (*p q*:*RIS*) := ∀ *I J*:*RI*, *p I* → *q J* → *RIlt I J*.

Definition *RISlc* (*p q*:*RIS*) := ∀ *I J*:*RI*, *p I* → *q J* → *RIlc I J*.

Definition *RISsum* (*p q*:*RIS*) (*K*:*RI*) := ∀ *I J*:*RI*,
p I → *q J* → *K* == (*I* + *J*).

Definition *RISdifference* (*p q*:*RIS*) (*K*:*RI*) :=
∀ *I J*:*RI*, *p I* → *q J* → *K* == (*I* - *J*).

Definition *RISproduct* (*p q*:*RIS*) (*K*:*RI*) := ∀ *I J*:*RI*,
p I → *q J* → *K* == (*I*×*J*).

Definition *RISmagnitude* (*p*:*RIS*) (*K*:*RI*) := ∀ *I*:*RI*,
p I → *K* == *RImagnitude I*.

Definition *RISzero* (*K*:*RI*) := *K* == *RIzero*.

Definition *RISfine* (*p*:*RIS*) := ∀ *epsilon*:*R*, (*Rzero*¡*epsilon*)%*R* →
∃ *I*:*RI*, *p I* ∧ (*epsilon* ¡ *RIlength I*)%*R*.

A.6. **Limits and completeness.** Module Limits

Require Import *QArith*.
Require Import *QI*.
Require Import *R*.
Require Import *RI*.
Require Import *RIS*.
Record *NL* : Type := *NLmake* {
NLset : *RIS*;
NLfine : *RISfine NLset*;
NLconsistent : *RISc NLset NLset* }.

Definition *RIinQI* (*L*:*RI*)(*I*:*QI*) :=
∀ (*l*:*R*)(*J*:*QI*), *RIel l L* → (*Rset l*) *J* → *QIsubset J I*.

Definition *Lim* (*Lambda*:*NL*)(*I*:*QI*) :=
∃ *K*:*RI*, (*NLset Lambda*) *K* ∧ *RIinQI K I*.
Check Lim.

Theorem *completeness* : ∀ *Lambda*:*NL*,
QISfine (*Lim Lambda*) ∧ *QISc* (*Lim Lambda*) (*Lim Lambda*).

A.7. **Inverse function theorem.** Module IFTheorem

```
Require Import QArith.
Require Import QI.
Require Import R.
Require Import RI.
Require Import RIS.
Require Import Limits.
Open Scope R_scope.

Lemma notzero1 : ∀ x:R, Rzero¡x → not (Rzero ¬ x).

Lemma notzero2 : ∀ x y:R, not (Rzero ¬ x) → x≤y → not (Rzero ¬ y).

Section IFT.
  Variables (f:R→R)(a b L K:R)(nz1:Rzero¡L)(nz2:L≤K)(a_le_b:Rle a b).

  Fixpoint I (n':nat) {struct n'} : RI :=
    match n' with
      — (S n) ⇒ RImake a b a_le_b          — O ⇒ RImake a b a_le_b
    end.

  Theorem inverse :
    (∀ x y:R, a≤x → x≤y → y≤b →
      (L*(y-x))¡=(f y - f x) → (f y - f x)¡=(K*(y-x))) →
    ∃ g:R→R, ∀ z w:R, (f a) ≤ z → z ≤ w → w ≤ (f b) →
      (f (g z)) == z
      ∧ (Rquotient (w-z) K (notzero2 L K (notzero1 L nz1) nz2) ≤ g w - g z
        ∧ g w - g z ≤ Rquotient (w-z) L (notzero1 L nz1)).

End IFT.

Section define_one_and_two_and_three.
  Definition QIone := QImake 1 1 (Qle_refl 1) : QI.
  Open Scope Q_scope.
  Definition QItwo := QImake (2#1)%Q (2#1)%Q (Qle_refl (2#1)%Q) : QI.
  Definition QIthree := QImake (3#1)%Q (3#1)%Q (Qle_refl (3#1)%Q) : QI.
  Close Scope Q_scope.
  Definition QISone (K:QI) := (K == QIone)%QI.
  Definition QIStwo (K:QI) := (K == QItwo)%QI.
  Definition QISthree (K:QI) := (K == QIthree)%QI.

  Lemma QISone_fine : QISfine QISone.
  Lemma QISone_consistent : QISc QISone QISone.
  Lemma QIStwo_fine : QISfine QIStwo.
  Lemma QIStwo_consistent : QISc QIStwo QIStwo.
  Lemma QISthree_fine : QISfine QISthree.
  Lemma QISthree_consistent : QISc QISthree QISthree.

  Definition Rone := (Rmake QISone QISone_fine QISone_consistent).
  Definition Rtwo := (Rmake QIStwo QIStwo_fine QIStwo_consistent).
  Definition Rthree := (Rmake QISthree QISthree_fine QISthree_consistent).
End define_one_and_two_and_three.

Definition Rsquare (x:R) : R := Rproduct x x.
```

```
Section Rsquare_bounds.
  Variables L K a b x y:R.
  Hypotheses (nz1:Rzero¡L)(nz2:L≤K).
  Hypotheses (le_ax:a≤x)(le_xy:x≤y)(le_yb:y≤b).

  Lemma Rsquare_upper_bound : (L*(y-x))¡=(Rsquare y - Rsquare x).

  Lemma Rsquare_lower_bound : (Rsquare y - Rsquare x)¡=(K*(y-x)).
End Rsquare_bounds.

Definition Rsqrt (x:R) := inverse Rsquare.
Check Rsqrt.
```

A.8. **Proposed axiomatisation of the theories of intervals.** Module XField

```
Definition associative (A:Type)(op:A→A→A) :=
  ∀ x y z:A, op (op x y) z = op x (op y z).
Definition commutative (A:Type)(op:A→A→A) :=
  ∀ x y:A, op x y = op y x.
Definition trichotomous (A:Type)(R:A→A→Prop) :=
  ∀ x y:A, R x y ∨ x=y ∨ R y x.
Definition relation (A:Type) := A → A → Prop.
Definition reflexive (A:Type)(R:relation A) := ∀ x:A, R x x.
Definition transitive (A:Type)(R:relation A) :=
  ∀ x y z:A, R x y → R y z → R x z.
Definition symmetric (A:Type)(R:relation A) := ∀ x y:A, R x y → R y x.

Record interval (X:Set)(le:X→X→Prop) : Set :=
  interval_make {
    interval_left : X;
    interval_right : X;
    interval_nonempty : le interval_left interval_right
  }.

Record I (grnd:Set)(le:grnd→grnd→Prop) : Type := Imake {
  Icar := interval le;
  Iplus : Icar → Icar → Icar;
  Imult : Icar → Icar → Icar;
  Izero : Icar;
  Ione : Icar;
  Iopp : Icar → Icar;
  Iinv : Icar → Icar;
  Ic : Icar → Icar → Prop;      Iplus_assoc : associative Iplus;
  Imult_assoc : associative Imult;
    Iplus_comm : commutative Iplus;
  Imult_comm : commutative Imult;
  Iplus_0_l : ∀ x:Icar, Ic (Iplus Izero x) x;
  Iplus_0_r : ∀ x:Icar, Ic (Iplus x Izero) x;
  Imult_0_l : ∀ x:Icar, Ic (Imult Ione x) x;
  Imult_0_r : ∀ x:Icar, Ic (Imult x Ione) x;
  Iplus_opp_r : ∀ x:Icar, Ic (Iplus x (Iopp x)) (Izero);
  Imult_inv_r : ∀ x:Icar, ~(Ic x Izero) → Ic (Imult x (Iinv x)) Ione;
```

$Imult\_plus\_distr\_l : \forall\ x\ x'\ y\ y'\ z\ z'\ z''$,
$Ic\ x\ x' \rightarrow Ic\ y\ y' \rightarrow Ic\ z\ z' \rightarrow Ic\ z\ z'' \rightarrow$
$Ic\ (Imult\ (Iplus\ x\ y)\ z)\ (Iplus\ (Imult\ x'\ z')\ (Imult\ y'\ z''))$;
$Ilt : Icar \rightarrow Icar \rightarrow$ Prop;
$Ilc :=$ fun $(x\ y{:}Icar) \Rightarrow Ilt\ x\ y \vee Ic\ x\ y$;
$Isup : Icar \rightarrow Icar \rightarrow Icar$;
$Iinf : Icar \rightarrow Icar \rightarrow Icar$;
$Ilt\_trans : transitive\ lt$;
$Ilt\_trich : \forall\ x\ y{:}Icar,\ Ilt\ x\ y \vee Ic\ x\ y \vee Ilt\ y\ x$;
$Isup\_lub : \forall\ x\ y\ z{:}Icar,\ Ilc\ x\ z \rightarrow Ilc\ y\ z \rightarrow Ilc\ (Isup\ x\ y)\ z$;
$Iinf\_glb : \forall\ x\ y\ z{:}Icar,\ Ilc\ x\ y \rightarrow Ilc\ x\ z \rightarrow Ilc\ x\ (Iinf\ y\ z)$;
$Ic\_refl : reflexive\ Ic$;
$Ic\_sym : symmetric\ Ic$
}.

Definition $interval_set\ (X{:}$Set$)(le{:}X{\rightarrow}X{\rightarrow}$Prop$)$:=
$(interval\ le) \rightarrow$ Prop.
Check interval_set.
Check Ic.
Check interval.
Definition $consistent\ (X{:}$Set$)(le{:}X{\rightarrow}X{\rightarrow}$Prop$)(TI{:}I\ le)(p{:}interval_set\ le)$
$:= \forall\ I\ J{:}interval\ le,\ p\ I \rightarrow p\ J \rightarrow Ic\ (i{:=}TI)\ I\ J$.
Check consistent.
Check Izero.

Definition $fine\ (X{:}$Set$)(zero{:}X)(minus{:}X{\rightarrow}X{\rightarrow}X)(le{:}X{\rightarrow}X{\rightarrow}$Prop$)(TI{:}I\ le)(p{:}interval_set\ le)$
$:= \forall\ epsilon{:}X,\ le\ zero\ epsilon \rightarrow$
$\exists\ I{:}(Icar\ TI),\ p\ I$
$\wedge\ le\ epsilon\ (minus\ (interval\_right\ I)\ (interval\_left\ I))$.

Record $N\ (grnd{:}$Set$)(zero{:}grnd)(minus{:}grnd{\rightarrow}grnd{\rightarrow}grnd)(le{:}grnd{\rightarrow}grnd{\rightarrow}$Prop$)(grndI{:}I\ le)$: Type := $Nmake$ {
$Ncar := interval\_set\ le$;
$Nconsistent := consistent\ grndI$;
$Nfine := fine\ zero\ minus\ grndI$;
$Nplus : Ncar \rightarrow Ncar \rightarrow Ncar$;
$Nmult : Ncar \rightarrow Ncar \rightarrow Ncar$;
$Nzero : Ncar$;
$None : Ncar$;
$Nopp : Ncar \rightarrow Ncar$;
$Ninv : Ncar \rightarrow Ncar$;
$Nc : Ncar \rightarrow Ncar \rightarrow$ Prop; $Nplus\_assoc : associative\ Nplus$;
$Nmult\_assoc : associative\ Nmult$;
$Nplus\_comm : commutative\ Nplus$;
$Nmult\_comm : commutative\ Nmult$;
$Nplus\_0\_l : \forall\ x{:}Ncar,\ Nc\ (Nplus\ Nzero\ x)\ x$;
$Nplus\_0\_r : \forall\ x{:}Ncar,\ Nc\ (Nplus\ x\ Nzero)\ x$;
$Nmult\_0\_l : \forall\ x{:}Ncar,\ Nc\ (Nmult\ None\ x)\ x$;
$Nmult\_0\_r : \forall\ x{:}Ncar,\ Nc\ (Nmult\ x\ None)\ x$;

Nplus_opp_r : $\forall$ *x*:*Ncar*, *Nc* (*Nplus x* (*Nopp x*)) (*Nzero*);
Nmult_inv_r : $\forall$ *x*:*Ncar*, ˜(*Nc x Nzero*) $\rightarrow$ *Nc* (*Nmult x* (*Ninv x*)) *None*;
Nmult_plus_distr_l : $\forall$ *x x' y y' z z' z"*,
Nc x x' $\rightarrow$ *Nc y y'* $\rightarrow$ *Nc z z'* $\rightarrow$ *Nc z z"* $\rightarrow$
Nc (*Nmult* (*Nplus x y*) *z*) (*Nplus* (*Nmult x' z'*) (*Nmult y' z"*));
Nlt : *Ncar* $\rightarrow$ *Ncar* $\rightarrow$ Prop;
Nlc := fun (*x y*:*Ncar*) $\Rightarrow$ *Nlt x y* $\vee$ *Nc x y*;
Nsup : *Ncar* $\rightarrow$ *Ncar* $\rightarrow$ *Ncar*;
Ninf : *Ncar* $\rightarrow$ *Ncar* $\rightarrow$ *Ncar*;
Nlt_trans : *transitive lt*;
Nlt_trich : $\forall$ *x y*:*Ncar*, *Nlt x y* $\vee$ *Nc x y* $\vee$ *Nlt y x*;
Nsup_lub : $\forall$ *x y z*:*Ncar*, *Nlc x z* $\rightarrow$ *Nlc y z* $\rightarrow$ *Nlc* (*Nsup x y*) *z*;
Ninf_glb : $\forall$ *x y z*:*Ncar*, *Nlc x y* $\rightarrow$ *Nlc x z* $\rightarrow$ *Nlc x* (*Ninf y z*);
Nc_refl : *reflexive Nc*;
Nc_sym : *symmetric Nc*
}.

References

[1] Milad Niqui. Formalising Exact Arithmetic: Representations, Algorithms and Proofs. PhD Thesis, Radboud University Nijmegen, September 2004.

[2] L. Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In H. Geuvers and F. Wiedijk, editors, Types for Proofs and Programs, volume 2646 of LCNS, pages 108–126. Springer-Verlag, 2003.

[3] Russell O'Connor. Few Digits 0.5.0.

[4] Gabriel Stolzenberg. A New Course of Analysis. Unpublished.