Cornell University

CS 4860 Final Project

The Constructive Reals

Zachary Zimmerman
ztz3@cornell.edu

Robert Constable
rc@cs.cornell.edu

December 6, 2016

1 Introduction

In class we learned about L.E.J Brouwer and his constructivist ideas, especially those relating to real analysis. We also learned how Errett Bishop brought those ideas to computing and gave "numerical meaning to as much as possible of classical abstract analysis" (Bishop, 3). I wanted to explore this further with a hands-on approach by implementing Bishop's ideas myself, with hope of gaining a deeper understanding of the practicality of constructivism in computing. It is with this motivation that I undertook the following journey to construct the real numbers in OCaml.

We are starting with the Integers $\mathbb{Z}$ and Rationals $\mathbb{Q}$ as implemented by the OCaml library Zarith. They are able to be stored and operated on with arbitrary precision, which we will assume correct for the purposes of this paper.

2 Motivation

Computer programs and systems widely utilize floating-point arithmetic for important calculations. However, these calculations do not guarantee accuracy. Floating-point numbers are represented as a significand and an exponent, allowing for representation of a much wider range of values as well as a wider range of precisions. Due to this representation, though, operations on floating point numbers often incur rounding and other errors.

For example, on February 25, 1991, the US radar system in Dhahran, Saudi Arabia failed to intercept a missile aimed for the base, which subsequently killed 28 soldiers. The failure was due to the accumulation of floating point errors which over the course of 100 hours of operation caused the internal clock to drift by one-third of a second. This drift lead the missile tracking system to miscalculate the location of the enemy missile by about 600 meters.[1]

2.1 Comparison

An example of a simple floating-point error is demonstrated below in OCaml. We perform the calculation $\left(\frac{1}{10^{14}} + 100\right)$ which results in an error of 4×10^{-15}.

```
utop # let a = 0.00000000000001;;
val a : float = 1e-14
utop # let b = 100.0;;
val b : float = 100.
utop # a +. b;;
- : float = 100.000000000000014
```

[1] from http://www.math.umn.edu/~arnold/disasters/patriot.html

However, performing the same calculation with the constructive reals yields the exact value to as many decimal places as desired.

```
utop # let a = R.inv (R.of_int 100000000000000);;
val a : '_a -> Q.t = <fun>
utop # let b = R.of_int 100;;
val b : '_a -> Q.t = <fun>
utop # (a+b) =~ 15;; (* calculates a+b to 15 decimal places *)
- : string = "100.000000000000010"
```

The advantages of arbitrary precision real numbers are clear, however, how can we construct them for use in computation?

3 Definition of Reals

We follow Bishop's definition of a real number[2]:

Definition 1. A sequence (x_n) of rational numbers is *regular* if

$$|x_m - y_n| \leq \frac{1}{m} + \frac{1}{n}$$

A *real number* is a regular sequence of rational numbers.

Further, Bishop continues:

> The rational number x_n is called the n^{th} *rational approximation* to the real number $x \equiv (x_n)$.

Essentially, the intuition is that each successive rational approximation x_n gets closer and closer to the "real" value of x. This gives us motive to formalize exactly how close we are to this ideal given a value of n. We consider this by taking the limit as $m \to \infty$ to find the precision.

$$|x - y_n| = \lim_{m\to\infty} |x_m - y_n| \leq \lim_{m\to\infty} \left(\frac{1}{m} + \frac{1}{n}\right) = \frac{1}{n}$$

Therefore the n^{th} rational approximation is always within $\frac{1}{n}$ of the true value.

4 Constructing Rationals

In OCaml, we create a module `R` (to represent $\mathbb{R}$) containing the type `t` which is a function from the integers to the rationals (`t` $= \mathbb{Z} \to \mathbb{Q}$). To start, we populate it with a few constants, such as `zero` and `one`.

[2] Constructive Analysis Chapter 2, pg. 18

```
module R =
    struct
      type t = Z.t -> Q.t
      let zero :t = function n -> Q.zero
      let one  :t = function n -> Q.one
    end;;
```

These values are trivial to implement since they are already rational, so for any value of n their approximation is exactly equal to their true value. And indeed this lets us easily construct reals from any rationals:

```
let of_int      (x:int) :t = function n -> Q.of_int x
let of_bigint   (x:Z.t) :t = function n -> Q.of_bigint x
let of_rational (x:Q.t) :t = function n -> x
```

However, how can we construct ourselves an irrational number from this framework?

5 Constructing an Irrational Number

We will attempt to construct an irrational number, e, that obeys Bishop's condition regarding accuracy, so that we might use it in our experiments with the constructive reals.

From the definition of e as $\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$, using the binomial theorem we arrive at the equivalent expression

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

From this, we might take the n^{th} approximation to be

$$e_n = \sum_{i=0}^{n} \frac{1}{i!}$$

To show the sequence (e_n) is regular, we compare two arbitrary approximations where $m > n$:

$$|e_m - e_n| = \left| \sum_{i=0}^{m} \frac{1}{i!} - \sum_{i=0}^{n} \frac{1}{i!} \right| = \left| \sum_{i=n}^{m} \frac{1}{i!} \right| \leq \frac{1}{m!} + \frac{1}{n!} \leq \frac{1}{m} + \frac{1}{n}$$

And so it follows that the approximation e_n will be accurate to e to within $\frac{1}{n}$.

5.1 In Practice

To find a more precise bound[3] on the error of rational approximations of e, we consider evaluating an arbitrary approximation e_N. For this value, the error is simply all the terms of the summation not accounted for:

$$|e - e_N| = \sum_{i=N+1}^{\infty} \frac{1}{i!} = \frac{1}{(N+1)!} + \frac{1}{(N+2)!} + \frac{1}{(N+3)!} \cdots$$

by expanding out the factorial slightly we can factor out $\frac{1}{N!}$ as follows

$$|e - e_N| = \frac{1}{N!}\left(\frac{1}{N+1} + \frac{1}{(N+1)(N+2)} + \frac{1}{(N+1)(N+2)(N+3)} + \ldots\right)$$

and because $k^2 < k(k+1)$ for positive integer k, we have

$$|e - e_N| < \frac{1}{N!}\left(\frac{1}{N+1} + \frac{1}{(N+1)^2} + \frac{1}{(N+1)^3} + \ldots\right)$$

which is simplified to the following using the formula for the sum of an infinite geometric series

$$|e - e_N| < \frac{1}{N!} \cdot \frac{1}{1 - \frac{1}{N+1}} = \frac{1}{N!} \cdot \frac{N+1}{N} < \frac{2}{N!}$$

To save on computation time, it is advantageous to calculate the smallest effective value of n which still guarantees sufficient accuracy. So to guarantee an accuracy of less than $\frac{1}{n}$, we only need to evaluate e to term N where $\frac{2}{N!} < \frac{1}{n}$. This means we can select N to satisfy $2n < N!$. Using the inverse to the factorial function (`ifact`), we find that $N = \lceil \texttt{ifact}(2n) \rceil$ will suffice.

5.2 Implementation of e

The implementation of e in OCaml follows directly from the relation between n and the expression for its n^{th} approximation.

```
let e :t = function n -> summation Z.zero (ifact (Z.mul ~$2 n))
  (function i -> Q.(inv (of_bigint (factorial i))))
```

This uses helper functions for `summation`, `factorial`, and `ifact` (which is written to round up already).

```
let rec summation first last f = if Z.gt first last then Q.zero else
  Q.add (f first) (summation (Z.succ first) last f);;
let rec factorial x = Z.(if leq x zero then one else
  mul x (factorial (pred x)));;
```

[3] Thanks to Mark Bickford for help in this section

```
let ifact x =
  let rec ifactrec bound p k = if Z.lt bound p then k else
    ifactrec bound Z.(mul p (succ k)) (Z.succ k) in
  ifactrec x Z.one Z.one
```

Also to note is that the `~$` operator is defined as a shorthand for casting expressions of type `int` into integers.

With this definition, we are able to represent the value of the real number e as a function in OCaml which can subsequently be evaluated to whatever level of accuracy that is desired[4].

6 Operations

Now that we have constructed real numbers, we must be able to operate on them in a way which preserve the accuracy guarantee from Bishop's definition.

6.1 Addition

The addition operator is perhaps the simplest of the binary operators to implement. It is almost nearly just the sum of the rational approximations of each input given a desired accuracy. The only tricky bit is that in order for the accuracy of the result to remain within $\frac{1}{n}$, the inputs must be evaluated to their $2n^{\text{th}}$ approximation.

```
let add (a:t) (b:t) :t = function n ->
  let two_n = Z.mul ~$2 n in
  Q.add (a two_n) (b two_n)
```

This results in the following accuracy guarantee:

$$\begin{aligned}
|\texttt{sum}(a,b)_m - \texttt{sum}(a,b)_n| &= |(a_{2m} + b_{2m}) - (a_{2n} + b_{2n})| \\
&= |(a_{2m} - a_{2n}) + (b_{2m} - b_{2n})| \\
&\leq |(a_{2m} - a_{2n})| + |(b_{2m} - b_{2n})| \\
&\leq \left(\frac{1}{2m} + \frac{1}{2n}\right) + \left(\frac{1}{2m} + \frac{1}{2n}\right) \\
&\leq \frac{1}{m} + \frac{1}{n}
\end{aligned}$$

[4] With the optimizations made to determine the least number of terms needed to calculate e to a given accuracy level, I was able to compute the value of e correctly to ten thousand decimal places in just a few minutes. It is expected that if more optimization techniques such as memoization are employed, the performance could be improved even further.

6.2 Multiplication

Similarly to addition, with multiplication the inputs must be evaluated to a higher precision than the desired output. The amount, however, relies on the magnitude of the greatest of the two inputs. To determine this, we create the `bound` helper function that determines the smallest integer larger than the magnitude of all of the approximations of its input. Because of the accuracy guarantee, we can determine this simply as $\texttt{bound}(x) = \lceil |x_1| + 2 \rceil$.

```
let mul (a:t) (b:t) :t = function n ->
  let bound x =
    let two  = Q.of_bigint ~$2 in
    Z.succ (Q.to_bigint (Q.add (Q.abs (x Z.one)) two)) in
  let k = max (bound a) (bound b) in
  let two_k_n = Z.mul (Z.mul ~$2 k) n in
  Q.mul (a two_k_n) (b two_k_n)
```

We also use the `max` function on integers, which is defined as below

```
let max a b = if Z.gt a b then a else b;;
```

This guarantees (since we know $|a_N| < k$ and $|b_N| < k$ for any N)

$$\begin{aligned}
|\texttt{mul}(a,b)_m - \texttt{mul}(a,b)_n| &= |a_{2km} \cdot b_{2km} - a_{2kn} \cdot b_{2kn}| \\
&= |a_{2km} \cdot b_{2km} - a_{2km} \cdot b_{2kn} + a_{2km} \cdot b_{2kn} - a_{2kn} \cdot b_{2kn}| \\
&= |a_{2km}(b_{2km} - b_{2kn}) + b_{2kn}(a_{2km} - a_{2kn})| \\
&\leq |a_{2km}|\,|b_{2km} - b_{2kn}| + |b_{2kn}|\,|a_{2km} - a_{2kn}| \\
&\leq k\left(\frac{1}{2km} + \frac{1}{2kn}\right) + k\left(\frac{1}{2km} + \frac{1}{2kn}\right) \\
&\leq \frac{1}{m} + \frac{1}{n}
\end{aligned}$$

6.3 Negation

Defining negation and proving it is trivial, as seen below

```
let neg (x:t) :t = function n -> Q.neg (x n)
```

$$|\texttt{neg}(a)_m - \texttt{neg}(a)_n| = |-a_m - -a_n| = |-1 \times (a_m - a_n)| = |a_m - a_n| \leq \frac{1}{m} + \frac{1}{n}$$

This gives us the ability to define subtraction in terms of negation and addition

```
let sub (a:t) (b:t) :t = add a (neg b)
```

6.4 Inversion

Inversion is much more difficult. It is important to avoid dividing by zero, and in order to do so, we must find an approximation of the real number which is farther than $\frac{1}{n}$ from zero, to ensure that the real number is separated from zero (i.e. not equal to zero). Note that this is not a decidable operation, so the program will not terminate if the input is equal to zero.

If we suppose the real number x is nonzero, then there exists some value N for which $|x_N| \geq \frac{1}{N}$, and also holds for all successive approximations. If we take the inverse of the $\max(nN^2, N^3)^{\text{th}}$ approximation to be the inverse of the real, we can use these facts to ensure the resultant real is accurate.

```
let inv (x:t) :t = function n ->
  let rec find_big_n i =
    if Q.geq (Q.abs (x i)) (Q.inv (Q.of_bigint i)) then i
    else find_big_n (Z.succ i) in
  let big_n = find_big_n n in
  if Z.lt n big_n then Q.inv (x (Z.pow big_n 3))
  else Q.inv (x (Z.mul n (Z.pow big_n 2)))
```

Take $j = \max(m, N)$ and $k = \max(n, N)$

$$\begin{aligned}
|\texttt{inv}(a)_m - \texttt{inv}(a)_n| &= \left|\frac{1}{a_{jN^2}} - \frac{1}{a_{kN^2}}\right| = \left|\frac{a_{jN^2} - a_{kN^2}}{a_{jN^2} \cdot a_{kN^2}}\right| = \frac{\left|a_{jN^2} - a_{kN^2}\right|}{\left|a_{jN^2} \cdot a_{kN^2}\right|} \\
&\leq \left(\frac{1}{jN^2} + \frac{1}{kN^2}\right) \frac{1}{\left|a_{jN^2}\right|} \cdot \frac{1}{|a_{kN^2}|} \\
&\leq \frac{1}{N^2}\left(\frac{1}{j} + \frac{1}{k}\right) N \cdot N \leq \frac{1}{j} + \frac{1}{k} \\
&\leq \frac{1}{m} + \frac{1}{n}
\end{aligned}$$

And from inversion, we can define division using multiplication

```
let div (a:t) (b:t) :t = mul a (inv b)
```

7 Field Axioms

The real numbers form a field under addition and multiplication, we will thus prove this for our construction in OCaml. To do so, we must verify the following properties hold true for our reals:

1. Closure under addition and multiplication. We need to show that for arbitrary reals a, b then $a + b$ is real and $a \times b$ is real.
2. Associativity under addition and multiplication. So $a+(b+c) = (a+b)+c$ and $a \times (b \times c) = (a \times b) \times c$.

3. Commutativity of addition and multiplication. $a + b = b + a$ and $a \times b = b \times a$
4. Existence of the additive identity. There exists the element 0 (`R.zero`) such that $a + 0 = a$ (`R.add a R.zero => a`).
5. Existence of the multiplicative identity. There exists an element 1 (`R.one`) such that $a \times 1 = a$ (`R.mul a R.one => a`)
6. Existence of additive inverse. For every element a there exists an element $-a$ such that $a + (-a) = 0$ (`R.add a (R.neg a)=> R.zero`).
7. Existence of the multiplicative inverse. For every element $a \neq 0$ there exists an element a^{-1} such that $a \times a^{-1} = 1$ (`R.mul a (R.inv a)=> R.one`)
8. Distributivity, $a \times (b + c) = a \times b + a \times c$.

7.1 Closure

Luckily enough, for closure the proof is in the program. Because the addition and multiplication functions are valid in OCaml, and have type signatures of `R.t -> R.t -> R.t`, these operations are guaranteed to have closure.

7.2 Associativity

Addition associative via the associativity of the rationals. One thing to note that comes up often is that simply operating on a real number causes its approximations to accelerate, getting more precise faster. However, because this is a pure inflation of n values, it is always an "improvement" in a sense.

```
(a + b) + c
=> function n ->
  Q.add ((function n -> (Q.add (a two_n) (b two_n))) two_n) (c two_n)
=> function n ->
  Q.add (Q.add (a four_n) (b four_n)) (c two_n)
=> function n ->
  Q.add (a four_n) (Q.add (b four_n) (b two_n))
=> a + (b + c)
```

```
(a * b) * c
=> function n ->
  Q.mul ((function n -> (Q.mul (a two_k_n) (b two_k_n))) two_k_n) (c
      two_k_n)
=> function n ->
  Q.mul (Q.mul (a four_k_sq_n) (b four_k_sq_n)) (c two_k_n)
=> function n ->
  Q.mul (a four_k_sq_n) (Q.mul (b four_k_sq_n) (c two_k_n))
=> a * (b * c)
```

7.3 Commutativity

Via commutativity of $\mathbb{Q}$ operations, trivially:

```
a + b
<=> function n -> Q.add (a two_n) (b two_n)
<=> function n -> Q.add (b two_n) (a two_n)
<=> b + a
```

```
a * b
<=> function n -> Q.mul (a two_k_n) (b two_k_n)
<=> function n -> Q.mul (b two_k_n) (a two_k_n)
<=> b * a
```

7.4 Identities

Similarly by the corresponding identities in $\mathbb{Q}$ we can prove these identities for the reals, however notice we also see precision acceleration here.

```
a + R.zero
=> function n -> Q.add (a two_n) (R.zero two_n)
=> function n -> Q.add (a two_n) ((function n -> Q.zero) two_n)
=> function n -> Q.add (a two_n) Q.zero
=> function n -> a two_n
=> a
```

```
a * R.one
=> function n -> Q.mul (a two_k_n) (R.one two_k_n)
=> function n -> Q.mul (a two_k_n) ((function n -> Q.one) two_k_n)
=> function n -> Q.mul (a two_k_n) Q.one
=> function n -> a two_k_n
=> a
```

7.5 Inverses

Negation is proven easily through evaluation

```
a + (R.neg a)
<=> function n -> Q.add (a two_n) ((R.neg a) two_n)
<=> function n -> Q.add (a two_n) ((function n -> Q.neg (a n)) two_n)
<=> function n -> Q.add (a two_n) (Q.neg (a two_n))
<=> function n -> Q.zero
<=> R.zero
```

With inversion there is some acceleration, here we assume a is far from zero:

```
a * (R.inv a)
=> function n -> Q.mul (a two_k_n) ((R.inv a) two_k_n)
=> function n -> Q.mul (a two_k_n) ((function n -> Q.inv (a n_N_sq))
    two_k_n)
```

```
=> function n -> Q.mul (a two_k_n) (Q.inv (a two_k_n_N_sq))
=> function n -> Q.one
=> R.one
```

7.6 Distributivity

Lastly, distributivity holds due yet again to the corresponding property for the rationals.

```
a * (b + c)
=> function n -> Q.mul (a two_k_n) ((b+c) two_k_n)
=> function n -> Q.mul (a two_k_n) ((function n -> Q.add (b two_n) (c
    two_n)) two_k_n)
=> function n -> Q.mul (a two_k_n) (Q.add (b four_k_n) (c four_k_n))
=> function n -> Q.add (Q.mul (a two_k_n) (b four_k_n)) (Q.mul (a
    two_k_n) (c four_k_n))
=> (a * b) + (a * c)
```

8 Decimal Accuracy

Suppose we want to evaluate a real number x to d number of decimal places. This will mean the error in our approximation of x should be not be enough to cause the d^{th} decimal to be any different. However, this cannot be guaranteed using a tolerance value since the value of x may have a string of nines or zeroes, creating the possibility for rounding errors. So, we choose to calculate a buffer of b extra decimal places (say, five or ten extra) in protection of this chance. The resultant tolerance goal becomes $\frac{1}{10^{d+b}}$, making $n = 10^{d+b}$. (Note: in the following code, we use $b = 1$ for efficiency)

```
let to_decimal (x:t) (digits:int) : string =
  let n = powZ ~$10 Z.(add (of_int digits) ~$1) in
  let characteristic = Q.to_bigint (x n) in
  let size_of_digits = powQ (Q.of_int 10) (Z.of_int digits) in
  let scaled_up = Q.(to_bigint (mul (x n) size_of_digits)) in
  let sign = if Z.(equal characteristic zero) && Z.(lt scaled_up zero)
    then "-" else "" in
  let mantissa = if Z.(equal characteristic zero)
    then Z.abs scaled_up else Z.rem scaled_up
    (Z.mul characteristic (Q.to_bigint size_of_digits)) in
  let mantissa_str = Z.to_string mantissa in
  let padding =
    String.make (digits - (String.length mantissa_str)) '0' in
  sign^(Z.to_string characteristic)^"."^padding^mantissa_str
```

9 Equality

Lastly, let us explore the question of how to deal with equality in the constructive reals. First we notice there can be two types of equality, strong and weak.

Strong equality would be to say that for all n, $x_n = y_n$ and so $x = y$. However, we see the possibility of two real numbers approaching the same value and yet by slightly different approximations which are still within $\frac{1}{n}$ and are equally valid. An example of this would be the equality of a and $a + 0$, which we have shown to be accelerated in n due to the nature of the addition operator. To reconcile this situation, we define weak equality (using the operator $\sim$) as follows:

$$\forall n. |a_n - b_n| \leq \frac{2}{n} \longleftrightarrow a \sim b$$

9.1 Equivalence Relation

We now endeavor to prove the above definition to be an equivalence relation on the constructive reals. To do so we show:

1. Reflexivity, $a \sim a$.
2. Symmetry, $a \sim b \longleftrightarrow b \sim a$.
3. Transitivity, $(a \sim b) \wedge (b \sim c) \longrightarrow a \sim c$

9.2 Reflexivity

Trivially, since $a_n = a_n$ for all n, then $a_n - a_n = 0 \leq \frac{2}{n}$.

9.3 Symmetry

Weak equality is symmetric because of the absolute value in the definition, so

$$|a_n - b_n| = |-1 \times (b_n - a_n)| = |b_n - a_n| \leq \frac{2}{n}$$

9.4 Transitivity

Given that $a \sim b$ and $b \sim c$, we know that for all n

$$|a_n - b_n| \leq \frac{2}{n} \text{ and } |b_n - c_n| \leq \frac{2}{n}$$

Suppose it is not the case that $a \sim c$. Then there exists a value N for which $|a_N - c_N| > \frac{2}{N}$. We phrase this as

$$|a_N - c_N| = \frac{2}{N} + \epsilon, \text{ where } \epsilon > 0$$

However, from $a \sim b$ and $b \sim c$ we have (using $n = kN$)

$$\begin{aligned} |a_{kN} - c_{kN}| &= |a_{kN} - b_{kN} + b_{kN} - c_{kN}| \\ &\leq |a_{kN} - b_{kN}| + |b_{kN} - c_{kN}| \\ &\leq \frac{2}{kN} + \frac{2}{kN} \\ &\leq \frac{4}{kN} \end{aligned}$$

It follows that the real value of a is at most $\frac{6}{kN}$ from the real value of c, since for $n = kN$ an approximation is at most $\frac{1}{kN}$ from its approximation.

Similarly, for $n = N$ we know that the approximations of a and c are at most $\frac{1}{N} + \frac{1}{N} + \frac{6}{kN} = \frac{2}{N} + \frac{6}{kN}$ apart, giving us

$$\frac{2}{N} + \epsilon \leq \frac{2}{N} + \frac{6}{kN}$$

For a given value of $\epsilon > 0$, we can always choose a value of k sufficiently large such that this relation does not hold. We simply need to choose $k > \frac{6}{\epsilon N}$ to find a contradiction.

Therefore, by contradiction we find $a \sim c$.

10 Conclusion

By following Bishop's lead, I was able to effectively implement the constructive reals in OCaml so that they might be used in practical applications. Further, I was able to prove (albeit somewhat informally) the correctness of various operations on them, showing that they are not at all vulnerable to inaccuracies in computation as floating-point operations are. It is my hope that this work may be useful someday to allow computer programmers to compute with arbitrary precision with the same ease of use that floating-point provides.

The code developed for this paper is available at:
`https://github.com/Oneman2feet/constructive-reals`

References

[1] Bickford, Mark; "Constructive Analysis and Experimental Mathematics using the Nuprl Proof Assistant." March 2, 2016. http://www.nuprl.org/documents/Bickford/reals.pdf

[2] Bishop, Errett and Bridges, Douglas; Constructive Analysis. Grundlehren der mathematischen Wissenschaften. Springer-Verlag, Berlin, 1985. http://nuprl.org/MathLibrary/ConstructiveAnalysis/

[3] Constable, Robert L.; *Lecture 23: CS 4860*. Cornell University November 15, 2016. http://www.cs.cornell.edu/courses/CS4860/2016fa/Lecture_23.pdf

[4] CS 3110 Spring 2016, *Problem Set 3: Version 3*. March 17, 2016. http://www.cs.cornell.edu/courses/cs3110/2016sp/hw/3/writeup.pdf