CS 3110 Spring 2016
Problem Set 3
Version 1 (last modified March 5, 2017)

Overview

This assignment involves building the Real numbers in OCaml, which requires implementing the natural and rational numbers. We represent these mathematical concepts with OCaml's module system. We will be using the definitions from *Foundations of Constructive Analysis* by Errett Bishop namely chapter 2. While we aim to keep this writeup self-contained, it is also a good idea to use this text to supplement your understanding of real numbers. **This assignment must be done individually.**

Objectives

- Gain mathematical maturity and an intuitive understanding of the number system
- Write Ocaml code that makes extensive use of modules, functors, and signatures.
- Gain experience writing specifications
- Familiarize with propositions as types

Recommended reading

The following supplementary materials may be helpful in completing this assignment:

- Lectures 8 9 10 11 12
- The CS 3110 style guide
- Real World OCaml, Chapter 4,9,10
- Constructive Analysis, Chapter 2

What to turn in

Exercises marked [code] should be placed in `naturals.ml`, `fields.ml`, and will be graded automatically. Exercises marked [written] should be placed in `written.txt` or `written.pdf` and will be graded by hand.

Compiling and testing your code

For this problem set we have provided a Makefile and minimal test cases.

- To compile your code, run `make main`
- To test your code, run `make test`
- To remove files from a previous build, run `make clean`

Please note that any submissions that do not compile will get an immediate 10% deduction from their original grade.

Testing with utop

You are also provided with a `.ocamlinit` file, which will run when you open utop.

Once you have compiled your code, the `#load` commands will load in the modules/functions/values from the compiled bytecode. This will be more useful than using the `#use` command, which you are familiar with.

If you haven't compiled, or you haven't implemented certain parts of the .ml files, you may get errors when opening utop. These are just the result of running the `.ocamlinit` commands; you can ignore them.

Style with Modules

There are two ways to open Modules for a local scope. As an example, let's consider opening the module `List`.

- Using parentheses (useful for one liners)

  ex) `List.length (List.tl (List.map (fun x -> x) l))` can be written as

  `List.(length (tl (map (fun x -> x) l)))`

- Using a let statment (useful for a specific section)

  `let open List in ...`

Some final notes

Once again, for loops and while loops are not allowed, nor are imperative features such as refs or arrays

Unlike the previous assignments, you are not limited in using recursion or List module functions (unless otherwise specified). You are free to use any previously defined function in subsequent problems. You can freely make any functions rec or not rec; the same goes for helper functions. You can also include “external” helper functions, if they are within a struct (so no top-level external helper functions).

Warmup: Natural Numbers

The natural numbers, $0, 1, 2, ...$, can be structurally defined as follows:

i. 0 is a natural number, and

ii. if n is a natural number, $S(n)$ is a natural number.

We represent the natural numbers by the signature, `N`, below. The signature contains a zero value and succ function, as per the definition, and is extended with addition, multiplication, equals, and less than operations.

```
module type N = sig
  (** [t] represents a natural number. *)
  type t

  (** [zero] returns the [t] representing 0. *)
  val zero : t

  (** [succ x] returns the successor to [x] *)
  val succ : t -> t

  (** [eq a b] returns [a = b] *)
  val eq : t -> t -> bool

  (** [add a b] returns [a + b]. *)
  val add : t -> t -> t

  (** [mul a b] returns [a * b]. *)
  val mul : t -> t -> t

  (** [lt a b] returns [a < b] *)
  val lt : t -> t -> bool

  (** [make x] returns [Some x] if x>=0, [None] otherwise.*)
  val make : int -> t option

  (** [to_int x] returns the integer representation of x
   *  it is undefined if x cannot be represented as an integer *)
  val to_int : t -> int
end
```

Exercise 1: Naturals with ints.

[code] Implement `N` using `ints`, in a module called `IntNat`, in `naturals.ml`.

Exercise 2: Naturals with big_ints.

[code] The `ints` in OCaml are 32-bit integers, which can overflow. We can instead use Big Ints, provided by the `Big_int` module, which are arbitrary-precision integers. A list of all its functions can be found here: `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Big_int.html`. Look over the Big_int module and get familiar with it, we will be using this module extensively for this problem set. Then, implement `N` using the type `Big_int.big_int`, in a module called `BigIntNat`, also in `naturals.ml`.

The Real Number System

The first proof of the existence of irrational numbers is often attributed to Pythagoras. It is believed that he discovered them by studying pentagrams. However, It wasn't until the work of Georg Cantor in 1871 where a rigorous definition of the real numbers was first given. Calculus itself developed without a rigorous definition of real numbers available. Since then, multiple definitions of real numbers have been proposed. In this problem set, we will work with the definitions provided by Errett Bishop.
Before discussing the construction of the real numbers it is important to understand the task at hand and what the set of real numbers should contain. First, the real numbers should include all the rational numbers and have the $+$ and $*$ operators. Namely, the real number system that we construct should form a field. We say that a set along with two operations $(+, *)$ is a **field** if it satisfies the following properties,

1. Closure under addition and multiplication. Consider $a, b \in F$ where F is a field, then we know that $a + b \in F$ and $a * b \in F$. Intuitively speaking, if we are operating within a field, we should never get an element that in not included within that field

2. Associativity under addition and multiplication. Let $a, b, c \in F$ then $a + (b + c) = (a + b) + c$ and $a * (b * c) = (a * b) * c$.

3. Commutativity of addition and multiplication. Let $a, b \in F$ then $a + b = b + a$ and $a * b = b * a$

4. Existence of the additive identity. There exists the element 0 such that $\forall a, a + 0 = 0 + a = a$

5. Existence of the multiplicative identity. There exists an element 1 such that $\forall a, a * 1 = 1 * a = a$

6. Existence of additive inverse. For every element a there exists an element $-a$ such that $a + (-a) = 0$

7. Existence of the multiplicative inverse. For every element $a \neq 0$ there exists an element a^{-1} such that $a * a^{-1} = 1$

8. Distributivity, For every element a, b, c, $a * (b + c) = a * b + a * c$.

A correct construction of the Real number system must satisfy all of these axioms. In this problem set *you are not expected to prove anything about the Reals* but we expect that you have a basic understanding of their definition, and the operations that we define.

We will look at several examples of fields in this problem set, and represent a field with the following signature:

```
module type FIELD = sig
  (** [t] is the type of the element in the field. *)
  type t

  (** [zero] is the additive identity *)
  val zero : t

  (** [one] is the multiplicative identity *)
  val one : t

  (** [add a b] returns [a + b] *)
  val add : t -> t -> t

  (** [mul a b] returns [a * b] *)
  val mul : t -> t -> t

  (** [neg x] returns [-x], the additive inverse of x *)
  val neg : t -> t

  (** [inv x] returns [1/x], the multiplicative inverse of x
   *  precondition: x is not equal to 0 *)
  val inv : t -> t

  (** [num_of_big_int b] returns the field representation of Big Int [b]
  val num_of_big_int : big_int -> t

  (** [num_of_int i] returns the field representation of int [i] *)
  val num_of_int : int -> t
end
```

An ordered field is a field with a total ordering of its elements. We implement them with another signature, `OrderedField`, which includes the `FIELD` signature above.

```
module type OrderedField = sig
  include FIELD

  (** [cmp a b] returns an integer i, such that:
   *  if a is greater than b, then i>0
   *  if a is equal to b,     then i=0
   *  if a is less than b,    then i < 0
   *)
  val cmp : t -> t -> int
end
```

We provide you with an example implementation of this signature: the module `Floats` implemented with OCaml's `floats` that attempts to represent the real numbers, $\mathbb{R}$. Unfortunately, floats have limited precision, and operations on the floats are not guaranteed to be mathematically correct. Thus, while the `Floats` has signature `OrderedField`, it does not actually implement a mathematical field. This is supported by the fact that, were we to strictly interpret our function specifications, this implementation would not be valid–e.g. [Floats.add x y] does not necessarily equal the mathematical sum of x and y.

0.1 Rational Numbers

Before you implement the reals, you must implement the rational numbers. A rational number is a number of the form p/q, where p and q are integers and q is nonzero. 2/1, 4/2, and 3/7 are rational numbers; π and e are not.

All natural numbers are integers, all integers are rational numbers, all rational numbers are real numbers, and so on. The rational numbers are a field, so the signature `Q` for them, below, will include the `OrderedField` module:

```
module type Q = sig
  include OrderedField

  (** [make p q] returns [Some x] where [x] represents the
      rational [p/q] if [q <> 0] and [None] otherwise. *)
  val make : big_int -> big_int -> t option

  (** [eq a b] returns [a = b]. *)
  val eq : t -> t -> bool

  (** [sub a b] returns [a - b]. *)
  val sub : t -> t -> t

  (** [div a b] returns [a / b].
    * precondition: b is not zero *)
  val div : t -> t -> t

  (** [abs x] returns x if x is non-negetive and [neg x] otherwise. *)
  val abs : t -> t

  (** [ceil x] returns the smallest integer that is strictly
      greater than x *)
  val ceil : t -> t

  (** [max a b] returns [a] if a>=b , [b] otherwise. *)
  val max : t -> t -> t
  (** [gcd a b] returns the gcd of [a] and [b] *)
  val gcd : t -> t -> t

  (** [to_int x] returns (p,q) where x is a rational representing p/q
   *  it is undefined if p or q cannot be represented as integers *)
  val to_int : t -> int * int

  (** [to_string x] returns a string version of your rational
    * For testing: implementation is up to you *)
  val to_string : t -> string
end
```

Exercise 3.

[code] Implement a module `Rationals` for the rational numbers with the above signature. Once again, to avoid overflow errors, you must implement this module using Big Ints instead of `ints`: specifically, you are given that `Rational.t` is a tuple of `big_ints`, in which the tuple (p,q) represents rational number p/q. Your implementation must maintain the invariant that for every value v of type t returned from a function specified in the `Q` signature, v is in reduced form (hint: there is a `gcd_big_int` function in `Big_int`).

0.2 The Real Numbers

Exercise 4: Definition of Real Numbers.

[code] As you learned in class, a real number can be defined as an infinite sequence of rational numbers getting closer and closer together. The formal given to us by Bishop of a constructive real numbers is as follows.
Definition: Let, $\{x_i\}$ denote an infinite sequence of rational numbers (e.g. $\{q_1, q_2, q_3, ...\}$). We say that $\{x_i\}$ is *regular* if for all $m, n \in \mathbb{N}$,

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

We say that a real number $r \in \mathbb{R}$ is regular sequence of rational numbers $\{x_i\}$, i.e $r \equiv \{x_i\}$

When defining addition and multiplication, it is important that this definition holds. This definition of real numbers motivates our type definition for our Real module. Namely, a real number is nothing more than a function from the natural numbers to the rational numbers, i.e $r : \mathbb{N} \to \mathbb{Q}$.

TODO: Examine the unimplemented MakeReal functor we have provided. It is a functor that takes a module of signature `N` and a module of signature `Q`, and returns a module with the following signature:

```
sig
  include OrderedField

  (** [make_real_from_rational x] returns the
        representation of x as a real number *)
  val make_real_from_rational : Rat.t -> t

  (** [approx x n] returns a rational q such that |x-q|< 1/n *)
  val approx : t -> Nat.t -> Rat.t
end with type t = Nat.t -> Rat.t
```

Note the definition for the `type t` in the `MakeReal` functor in `fields.ml` that we have given you, and how that definition corresponds with the above definition of a real number. We specify that `type t = Nat.t -> Rat.t` in the signature to expose this type outside of its implementation.

TODO: Implement `approx: t -> Nat.t -> Rat.t` based on this information.

Exercise 5: Rationals as Reals, Multiplicative and Additive Identities.

[code] Surely, the field of Reals needs to contain rational numbers as well. This is done in the simplest way; we take a constant sequence. In other words, if $\frac{p}{q} \in \mathbb{R}$, then $\frac{p}{q} \equiv \{\frac{p}{q}\}$. The proof that this sequence satisfies the definition of rationals is trivial.
TODO: Implement the `make_real_from_rational: Rat.t -> t`, `one: t`, and `zero: t` functions in the `MakeReal` functor in `ps3.ml`.

Exercise 6: Definition of Addition, and Negation.

[code] Addition and Negation are done in the following way.
Definition: Let $x = \{x_n\}$ and $y = \{y_n\}$ be two real numbers. Then $x + y = \{x_{2n} + y_{2n}\}$. We claim that by adding real numbers this way will make the sequence a valid sequence. Indeed, let $z \equiv x + y$, then

$$\begin{aligned} |z_n - z_m| &= |x_{2n} + y_{2n} - x_{2m} - y_{2m}| \\ &\leq |x_{2n} - x_{2m}| + |y_{2n} - y_{2m}| \\ &\leq \frac{1}{2n} + \frac{1}{2m} + \frac{1}{2n} + \frac{1}{2m} = \frac{1}{n} + \frac{1}{m} \end{aligned}$$

Definition: Let $x \equiv \{x_n\}$ be a real number, its additive inverse is then given by $-x \equiv \{-x_n\}$. The proof of this is trivial. And we see that $x + (-x) = \{x_{2n} - x_{2n}\} = \{0\}$.

TODO: Given the above definitions, implement the `add: t -> t -> t` and `negate: t-> t` function in the `MakeReal` functor.

Exercise 7: Defining Multiplication.

[code] Defining multiplication is a little trickier. The reason is that defining multiplication the naive way, i.e $x * y = \{x_n y_n\}$ will break the regularity requirement. To define multiplication, we need an upper bound. That is for every $x \in \mathbb{R}$ we need some number K_x such that $\forall n \in \mathbb{N}$

$$|x_n| < K_x$$

Luckily this number can be found by finding the least integer which is greater than $|x_1| + 2$. We call K_x the *canonical bound* for x. Using this value, we can now define multiplication
Definition: Let $x \equiv \{x_n\}$ and $y \equiv \{y_n\}$ be real numbers with canonical bounds K_x, K_y respectively. Let $k = \max(K_x, K_y)$. Then,

$$x * y = \{x_{2kn} y_{2kn}\}$$

We claim that this definition of multiplication will satisfy the regularity of the sequence. Indeed, let x, y be real numbers and let $z = x * y$, then

$$|z_n - z_m| = |x_{2kn}y_{2kn} - x_{2km}y_{2km}| = |x_{2kn}y_{2kn} - x_{2kn}y_{2km} + x_{2kn}y_{2km} - x_{2km}y_{2km}|$$
$$\leq |x_{2kn}||y_{2kn} - y_{2km}| + |y_{2km}||x_{2kn} - x_{2km}|$$
$$\leq k(\frac{1}{2kn} + \frac{1}{2km}) + k(\frac{1}{2km} + \frac{1}{2kn}) = \frac{1}{m} + \frac{1}{n}$$

This shows that the regularity requirement is satisfied by multiplication.
TODO: Given the above definition of multiplication, implement the `multiply: t -> t -> t` function in the `MakeReal` functor

Exercise 8: Defining Inverses.

You are given the implementation if `inverse` in `fields.ml`! You DO NOT have to write your own implementation. However, if you are curious, the reasoning behind the provided implementation is given below.

Defining inverses is perhaps the trickiest operation. The reason for this is that the naive implementation $x^{-1} \equiv \{x^{-1}\}$ does not account for the fact that an element in the sequence might be zero. Luckily it a known fact that if $x \neq 0$ then it will have at most a finite number of zeroes in its sequence. In other words, there exists a point in the sequence n such that $\forall m > n$.

$$|x_m| \geq N^{-1}$$

After some point every element in the sequence is bounded from below. Our challenge is to find that N. First, consider the following definition of a positive number.
Definition: A real number $x \equiv \{x_n\}$ is said to be positive if

$$x_n > n^{-1}$$

For some n. Now we can prove that positive real numbers are bounded from below. Therefore, for every positive real number x we can find some n such that the definition holds. We claim that if we let $N \in \mathbb{Z}$ satisfy,

$$\frac{2}{N} \leq x_n - n^{-1}$$

then $\forall m \geq N$,

$$x_m \geq N^{-1}$$

After finding N we are now able to define inverses as follows
Definition: Let x be a non-zero real number. There exists a positive integer N with $|x_m| \geq N^{-1}$ for $m \geq N$. Then we define the inverser $y = x^{-1}$,

$$y_n = (x_{N^3})^{-1} \ (n < N)$$

$$y_n = (x_{nN^2})^{-1} \ (n \geq N)$$

We assert that this construction will provide a regular sequence and thus is a well-defined real number. We recommend reading chapter 2 of Erett Bishop's book for a full proof.

Exercise 9: Implementing a comparison operator.

[code] Recall the definition of a regular sequence of rational numbers $\{x_i\}$:

$$|x_n - x_m| \leq \frac{1}{n} + \frac{1}{m}, \forall m, n \in \mathbb{N}$$

Thus, $\lim_{m\to\infty} |x_n - x_m| \leq \frac{1}{n}, \forall n \in \mathbb{N}$. Conceptually, this states that rational number x_n is at most $\frac{1}{n}$ away from the rational number that $\{x_i\}$ converges to. From this logic follows the definition of equality and the definition of a positive number, as stated before in exercise 8.
Definition: Two real numbers $x \equiv \{x_i\}, y \equiv \{y_i\}$ are *equal* if

$$|x_n - y_n| \leq \frac{2}{n}, \textbf{for all } n \in \mathbb{N}$$

Definition: A real number $x \equiv \{x_n\}$ is positive if

$$x_n > n^{-1}, \text{ for } \textbf{some } n$$

Lastly, from these definitions, we define greater and less than.
Definition: Let x and y be real numbers. Then,

$$x > y \text{ (or } y < x\text{) if } x - y \in \mathbb{R}^+$$

TODO: Given the above definitions of equality, a positive number, and greater and less than, implement the `cmp: t -> t -> t` function in the `MakeReal` functor.

Implementing the square root

Congratulations! Up to this point, you have built from scratch a powerful real number system supporting arbitrary $\epsilon = 1/n$ precision. To fully appreciate the awesomeness of this module, you will implement the square root function on the Reals, which will give you arbitrarily precise representation of an iconic irrational number, $\sqrt{2}$.

Exercise 10: Square Root of Reals.

[code] Consider a nonnegative real $x \equiv \{x_n\}$, i.e. $\forall n > 0, x_n \geq -n^{-1}$. This constraint is just to ensure the square root of x is indeed real.
Now the intuition of taking square root of a sequence of rationals is naturally to take square root of each element individually, but we don't know how to efficiently do that for an arbitrary rational. Fortunately we do know how to find the square root of a non-negative **integer**. In fact as you might have seen in class, finding the *int-sqrt* of an integer n can be done in $O(logn)$ time. Put more formally, given $n \in \mathbb{N}$, $m = isqrt(n)$ if $m \in \mathbb{N}, m^2 \leq n$ and $(m+1)^2 > n$.
TODO: Implement the helper function `isqrt` in `sqrt`.
To use *isqrt*, we need to approximate x_n by integers. This can be done by rewriting

$$x_n = p_n/q_n \approx a_n/2n \text{ for some } a_n \in \mathbb{Z}$$

a_n can be solved by the following formula. Note that here the division is integer division.

$$a_n = (p_n * 2n)/q_n$$

Now let's define $y \equiv y_n$ by

$$y_n = isqrt(a_n * 2n)/2n$$

It's easy to verify that y_n^2 approximates x_n by only the errors incurred by integer square root. Note that here there is indeed some caveats: we have actually introduced two types of errors. The error in approximating x_n with a_n and the error in approximating the integer square root of a_n. Fortunately both these errors can be simultaneously bound and y_n here is still regular, and in fact y is the square root of x. For those who are interested in the proof, please consult Mark Bickford's "Constructive Analysis and Experimental Mathematics using the NuPrl Proof Assistant".
TODO: Implement `sqrt: Reals.t -> Reals.t`
Now you can plug in some numbers built from rationals into your sqrt functions and approximate $\sqrt{2}$ to any precision you like!

List Sort and Functors

As we have implemented the `OrderedField` signature with several different modules, we can write functions that apply to all ordered fields, by only using the values defined by that signature. We do this by creating a functor `FieldFunctions`, which takes in a module `F : OrderedField` and returns a new module containing functions specific to the field module inputted. These new functions are specified by `FieldFunctions`'s signature:

```
sig
  (** Sorts a list with unique elements according to F.cmp *)
  val sort_list : F.t list -> F.t list

end
```

Exercise 11.

[code] Implement the sort_list function, which should sort a list that has only unique elements into increasing order using the ordered field's comparison function. Recall that comparison function will return 0 if its arguments compare as equal, a positive integer if the first is greater than the second, and a negative integer if the first is smaller.

```
  let rec sort_list (lst: F.t list) : F.t list = ...
```

For example,

```
  # Floats.FloatFunctions.sort_list [1.; 5.; 4.; 0.];;
  - : float list = [0.; 1.; 4.; 5.]
```

You can find a couple more examples in `test.ml`.

This function must be implemented by using recursion. You can use List module functions, with the exception of the list sorting functions (List.sort, List.stable_sort, etc. In other words, any functions under the "Sorting" banner in the List Module reference). Your sort algorithm must scale in a reasonable fashion with the list length.

Exercise 12.

[written] Briefly explain why sorting won't work with our definition of real numbers without the stipulation that the list contains only unique elements.

Specifications and Logic

Exercise 13.

[written] Write formal specifications (including preconditions(s), postcondition(s)) for the following functions:

i. `gcd`, in the signature `Q`

ii. `sqrt`, from exercise 10

iii. `sort_list`, from exercise 11

Exercise 14.

[written]Expand the following abbreviated logical formulas into their primitive definitions, e.g. $\sim \alpha$ to $\alpha \to Void$. Then give a program in the corresponding type and comment on how it provides evidence for the "computational truth" of the formula.

1. $\sim (\alpha \vee \beta) \Rightarrow \sim \alpha \,\&\, \sim \beta$
2. $(\alpha \Rightarrow \gamma) \,\&\, (\beta \Rightarrow \gamma) \Rightarrow (\alpha \vee \beta) \Rightarrow \gamma$
3. $(\alpha \Rightarrow \beta) \Rightarrow (\sim \beta \Rightarrow \sim \alpha)$

Comments

[written] At the end of the file, please include any comments you have about the problem set, or about your implementation. This would be a good place to document any extra Karma problems that you did (see below), to list any problems with your submission that you weren't able to fix, or to give us general feedback about the problem set.

Release files

The accompanying release file `ps3.zip` contains the following files:

- `writeup.pdf` is this file.
- `.ocamlinit` to help you with utop configurations.
- `release/naturals.ml`, `release/fields.ml` and `written.txt` are templates for you to fill in and submit.
- The .mli files contain the interface and documentation for the functions that you will implement in their respective .ml files