Programming Languages: Recitation 10

Goktug Saatcioglu

04.18.2019

1 HW8 Problem 1

The answers below are intuitive structure-based solutions to problem 1. Of course, these problems could have also been solved using the Curry-Howard correspondence which presents a slightly more methodical way of solving inhabitation problems.

1. We wish to implement a function that takes a a pair `('a * 'b)` and returns the first element of the pair. This one is straightforward and we have multiple correct implementations which are given below.

   ```
   let f1 (a,b) = a

   let f2 = fst

   let f3 = function
   | (a,b) -> a
   ```

2. This function signature should be read as follows: we have three arguments where the first is a function from type `'a` to type `'b`, the second argument is a function from type `'b` to type `'c`, the third argument is something of type `'a`, and the function returns something of type `'c`. So if we apply the first argument to the third we get something of type `'b` and then we can apply this result to the second argument to get something of type `'c`. The function is given below.

   ```
   let g x y z = y (x z)
   ```

3. This function signature should be read as follows: we have three arguments where the first is a function that takes a pair of type `'a * 'b` and returns a type `'c`, the second arugment is something of type `'a`, the third argument is something of type `'b`, and the function returns something of type `'c`. So if we take the second and third argument and turn it into a pair we will get something of type type `'a * 'b` which we can then apply to the first argument to get a result of type `'c` which is the answer wanted. The function is given below.

   ```
   let h f a b = f (a, b)
   ```

4. This function signature should be read as follows: we have three arguments where the first is a function that takes arguments of type `'a` and type `'b` to give a result of type `'c`, the second argument is a pair of type `'a * 'b`, and the function returns something of type `'c`. So if we retrieve the contents of the second argument, which is a pair, then we can apply the first and second elements of the pair to the first argument to get a result of type `'c`. The function is given below and notice that there are multiple correct answers.

   ```
   let i1 f (a,b) = f a b

   let i2 f p = f (fst p) (snd p)
   ```

```
let i3 f = function
| (a, b) -> f a b
```

5. We have the ADT either which is given below.

```
type ('a,'b) either =
| Left of 'a
| Right of 'b
```

The function signature should be read as follows: the first argument is a pair `(’a, ’b) either * (’a -> ’c) * (’b -> ’c)` meaning we have something of type `('a, 'b) either` as the first element of the pair which means that we can either get a `Left` whose contents are of type `'a` or a `Right` whose contents are of type `'b`, the second element of the pair is a function from type `'a` to `'c` and the third element of the pair is a function from type `'b` to `'c`, and the function returns something of type `'c`. So we know that we can match on the first element of the pair to either get a `Left` whose contents are of type `'a` or a `Right` whose contents are of type `'b` and then if we have a `Left` then we apply its contents to the second element of the pair to get a result of type `'c` and if we have a `Right` then we apply its contents to the third element of the pair to get a result of type `'c`. The function is given below and notice that there are multiple correct answers.

```
let j1 (e,f,g) = match e with
| Left (a) -> f a
| Right (b) -> g b

let j2 = function
| Left (a), f, _ -> f a
| Right (b), _, g -> g b
```

6. The function signature should be read as follows: the first argument is of something of type `(’a, ’b * ’c) either` meaning we can either encounter a `Left` whose contents are of type `'a` or a `Right` whose contents are a pair of type `'b * 'c`, and we wish to return a pair of `either`’s where the first element is of type `('a, 'b) either` meaning we either return a `Left` whose contents are of type `'a` or a `Right` whose contents are a pair of type `'b` and the second element is of type `('a, 'c) either` meaning we either return a `Left` whose contents are of type `'a` or a `Right` whose contents are a pair of type `'c`. So we must consider what we can possibly see as input and then return the appropriate outputs given the typing information. If we see a `Left` then we know its contents are of type `'a` so we can construct a pair of `either`’s from something of type `'a` only. If we return the pair `Left` of type `'a` and `Left` of type `'a` then we get a result of type `('a, 'b) either * ('a, 'c) either` which is what we wanted. If we see a `Right` then we know that its contents are of type `'b * 'c` and we seek to get a result type `('a, 'b) either * ('a, 'c) either`. So then the first element must be `Right` of type `'b` and the second element must be `Right` of type `'c` meaning we can extract the elements of the pair in our input and return the appropriate pair. Collecting everything together, we get the solution given below.

```
let k = function
| Left (a) -> Left (a), Left (a)
| Right (b, c) -> Right (b), Right (c)
```

2 HW8 Problem 2

The answers below skip over the steps required to derive the most general unifier but go over how to obtain all type equality constraints derived from the body of the functions. This is because it is rather easy to solve the constraints once the correct constraints are found but the task of discovering all the constraints can be difficult. So we focus on that aspect. Furthermore, the most general unifiers are already available in the online answer key so we omit the solutions here but rather talk about the intuition why the functions are well-typed or do not type. To also see the most general unifier please see the solutions the HW8 on the class GitHub page. Do make sure to still go over each step the constraint solving so as to either observe a contradiction or a most general unifier to make sure you know the material well enough. Furthermore, if you ever get stuck on a type inference question you can always derive the constraints from the AST and then solve them to get the correct answer which gives you a systematic way to solve such questions.

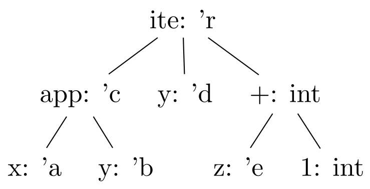
1. We have a function definition of the form

   ```
   let f x y z = if x y then y else z + 1
   ```

   and we begin by giving names to the function arguments

   ```
   'f = 'x -> 'y -> 'z -> 'r
   ```

   where `'r` is the returned type of the function. Then we draw the AST of the function body where we label each node with a new type name so as to label each sub-expression with a new type name.

   Note how `+` and `1` are labelled as `int` as their types are not polymorphic. Furthermore, the root of the tree is labelled with type `'r` because this is what the function returns. Starting from the leaves of the tree we can then derive type constraints which gets us

   ```
   'a = 'x
   'b = 'y
   'e = 'z
   ```

   where these are derived by looking at the function signature we named above. Then we move one level up the tree and consider the function application and the addition. We can conclude that `x` must take something of type `'b` and return something of type `'c`. Addition on the other hand tells us that `z` must be of type `int`. This gives us the additional constraints given below.

   ```
   'a = 'b -> 'c
   'e = int
   ```

   Finally, we move one more level above and notice we have an if-then-else statement so the result of the if branch must be a `bool` and the then and else branches must return the same type which is equivalent to what the ite itself returns. So we get three more constraints which are given below.

```
'c = bool
'd = 'r
int = 'r
```

These are all the constraints we need to show that the function will type. Intuitively, can you explain why the function is well-typed? The idea is that since the else branch uses an addition then `z` must be of type `int` and the else branch returns an `int`. Since an if-then-else must return the same type for both branches then `y` must also be of type `int`. Furthermore, since an if conditional must always be of type `bool` the condition `x y` must take an `int` as `y` is an `int` and return a `bool`. There are no contradictions as we reason about the type information so we know that the function is well-typed. The signature is given below.

```
f : (int -> bool) -> int -> int -> int
```

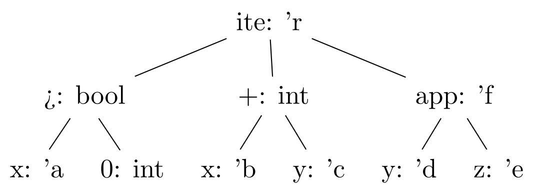
2. We have a function definition of the form

```
let g x y z = if x > 0 then x + y else y z
```

and we begin by giving names to the function arguments

```
'g = 'x -> 'y -> 'z -> 'r
```

where `'r` is the returned type of the function. Then we draw the AST of the function body where we label each node with a new type name so as to label each sub-expression with a new type name.



Note how `>`, `0` and `+` are labelled as `bool`, `int` and `int` respectively as their types are not polymorphic. Furthermore, the root of the tree is labelled with type `'r` because this is what the function returns. Starting from the leaves of the tree we can then derive type constraints which gets us

```
'a = 'x
'b = 'x
'c = 'y
'd = 'y
'e = 'z
```

where these are derived by looking at the function signature we named above. Then we move one level up the tree and consider the comparison, the addition and the function application and the addition. We can conclude that `y` must take something of type `'d` and return something of type `'e`. Addition on the other hand tells us that `x` and `y` must be of type `int`. Also, comparison tells us that since the right hand side operand is of type `int` the left hand operand must be of type `int` because we can only compare the same types meaning `x` must be of type `int`. This gives us the additional constraints given below.

```
'a = int
'b = int
'c = int
'd = 'e -> 'f
```

Finally, we move one more level above and notice we have an if-then-else statement so the result of the if branch must be a `bool` which is trivially true in this case and the then and else branches must return the same type which is equivalent to what the ite itself returns. So we get two more constraints which are given below.

```
int = 'r
'f = 'r
```

These are all the constraints we need to show that the function will not type. Intuitively, can you explain why the function is not well-typed? The derived constraints reveals everything. We concluded that the `y` in `x + y` must be of type `int`. Furthermore, the `y` in `y z` must be of type `'z -> 'r` which becomes `'z -> 'int` as the if-then-else branches must return the same types and the then branch returns an `int` due to addition. So we try to find the most general unifier for these two possible types and realize it is impossible to find one. The variable `y` cannot be both a function and an `int` constant at the same type so we can conclude the function does not type.

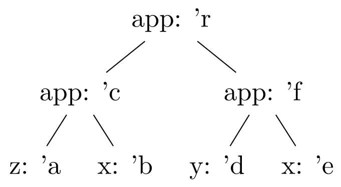
3. We have a function definition of the form

```
let h x y z = z x (y x)
```

and we begin by giving names to the function arguments

```
'h = 'x -> 'y -> 'z -> 'r
```

where `'r` is the returned type of the function. Then we draw the AST of the function body where we label each node with a new type name so as to label each sub-expression with a new type name.



We must take care to get the correct AST and the key is to note that function application is left associative. So the expression `z x (y x)` should be read as `(z x) (y x)` which helps us get the correct AST. Furthermore, the root of the tree is labelled with type `'r` because this is what the function returns. Starting from the leaves of the tree we can then derive type constraints which gets us

```
'a = 'z
'b = 'x
'd = 'y
'e = 'x
```

where these are derived by looking at the function signature we named above. Then we move one level up the tree and consider the two function applications. Since at this point we know how to infer the constraint of a function application we skip the details. This gives us the additional constraints given below.

```
'a = 'b -> 'c
'd = 'e -> 'f
```

Finally, we move one more level above to the root and notice we have another function application. This time the application must return a type `'r` as this is what the function return but otherwise everything else is standard. So we get one more constraint which are given below.

```
'c = 'f -> 'r
```

Note that we can the result of a function application to the result of another function application and this is known as currying (so don't let the somewhat odd root level of the AST throw you off). In fact, a function application such as

```
(fun x y -> x + y) 1 2
```

is actually

```
(fun x -> fun y -> x + y) 1 2
```

so that the application of the first argument leads to another function

```
(fun y -> 1 + y) 2
```

which we then apply `2` to get the result. Now that we have all the constraints we need to show that the function will type. Intuitively, can you explain why the function is well-typed? This time it is a little more complicated but nonetheless we can figure it out by just looking at the function body. Let's first consider the term `(y x)`. This means that `y` is a function that takes something of type `'x` and returns something arbitrary such as of type `'y`. Now let's consider the term `z x (y x)` which is actually read as `(z x) (y x)` which in turn hints to us that `z` is a function that takes two arguments. The first argument is simply `x` so we can say it takes a type `'x` and this will be consistent with our typing for `(y x)`. The second argument is of type `(y x)` which returns a type `'y` so we can say the second argument `z` should accept should also be of type `'y`. Again, we observe no contradictions so everything is fine. Finally, the function `z` should return something and since there are no restrictions in the body on what it should return it can return some type `'z`. Overall, we see no contradictions and manage to intuitively unify the statement so that we can conclude the function is well-typed. The signature is given below.

```
h : 'x -> ('x -> 'y) -> ('x -> 'y -> 'z) -> 'z
```

3 Using OCaml's Map Functor

- The OCaml standard library provides an implementation of dictionary data structure using balanced binary trees. You can see the implementation here and its signature here.

- "The Map module defines a functor `Make` that creates a structure implementing a map over a particular type of keys. That type is the input structure to `Make`. The type of that input structure is `Map.OrderedType`, which are types that support a compare operation:"

  ```
  module type OrderedType = sig
    type t
    val compare : t -> t -> int
  end
  ```

  The reason we need compare is because th internal representation uses balanced binary trees that require comparisons.

- We can create a mapping from integers to integers as follows. First we create a module that is of type `OrderedType`. The type `t` needs to be `int` as this is what we would like our keys to be. Then we define comparison as being the normal way OCaml compares two `int`'s and this is what the function `Pervasives.compare` does.

  ```
  module IntKeys =
    struct
      type t = int
      let compare = Pervasives.compare
    end
  ```

  Then we can use `Map.Make` to create a mapping of integers to values.

  ```
  module IntMap = Map.Make(IntKeys)
  ```

  Now let's use our map and carry out some of the functions that are available to us.

  ```
  let m = IntMap.(empty |> add 1 2 |> add 4 5)
  let i = IntMap.(m |> find 1)
  let i' = IntMap.(m |> remove 1 |> find 1)
  ```

  Note that we can also have a mapping from integers to strings as long as all the values have type string or we can define a new module `StringKeys` to have a mapping from strings to some types (can be int and so on).

- A list of Map functions can be found here and a very short tutorial here. Note that the same principles above also apply to the Set module so you should be able to create those too.

- As an additional note, we can also create our `IntMap` in a terser way. This is show below.

  ```
  module IntMap = Map.Make(struct type t = int let compare = Pervasives.compare end)
  ```

- You can also find some nice notes on modules and functors in OCaml here.

4 Modules and Functors

- During lecture we learned about OCaml's module system and how we can use modules to paramaterize over other modules (functors). We will now implement our own OCaml module for arbitrary precision arithmetic and then use it to implement another module for arbitrary precision rational arithmetic.

- For the sake of brevity (to finish the exercise during recitation), we will consider only arbitrary precision natural numbers. That is we would like to have a module that does arithmetic on arbitrarily large positive numbers such as those larger than `max_int`. We will limit our module to the arithmetic operators `+`, `-` and `*`. Furthermore, we would like a function that converts any arbitrary number to an integer or string (as our numbers may overflow) and we would like a function that takes an integer and converts to our representation of natural numbers. Consider the following module signature.

```
(** Signature of module providing arbitrary precision arithmetic on natural numbers *)
module type BigNatType =
  sig
    type bignat

    (** The number 0 **)
    val zero: bignat

    (** The number 1 **)
    val one: bignat

    (** Convert an int to a big natural *)
    val from_int : int -> bignat

    (** Convert a big natural to an int (may overflow) *)
    val to_int : bignat -> int

    (** Convert a big naturals to a string in decimal representation *)
    val to_string : bignat -> string

    (** Addition of big naturals *)
    val (+) : bignat -> bignat -> bignat

    (** Subtraction of big naturals. Raises [Invalid_argument] if result is negative. *)
    val (-) : bignat -> bignat -> bignat

    (** Multiplication of big naturals *)
    val ( * ) : bignat -> bignat -> bignat
  end
```

  Our module `BigNat` uses a type `bignat` which is the type of our representation of natural numbers. However, we do not specify the concrete type here so as to hide the information from a client that may use the module. Remember that this is an important feature of modules, as it creates a barrier between the implementation of a module and its client code that prevents the client code from breaking when the implementation of the module changes. This is critical for the design and maintenance of large code bases as changes to the code base should be insulated from affecting large portions of the code. Furthermore, we also have defined the functions `zero` and `one` to abstract away the two numbers required to define arithmetic on natural numbers.

- Let's now begin implementing this module. Consider the partially completed implementation of the `BigNat` module given below.

```
module BigNat : BigNatType =
  struct
    type bignat = int list (* least significant first *)

    let base = 10000 (* must satisfy: float_of_int base <= sqrt (float_of_int max_int) *)
```

```ocaml
let zero = []

let one = [1]

let rec from_int = function
  | 0 -> []
  | n ->
      if n < 0 then raise (Invalid_argument "negative number")
      else n mod base :: from_int (n / base)

let rec to_int = function
  | [] -> 0
  | d :: ds -> d + base * to_int ds (* may overflow *)

let rec pad =
  let base_len = String.length (string_of_int base) - 1 in
  fun s -> if String.length s = base_len then s else pad ("0" ^ s)

let to_string ds =
  match List.rev ds with
  | [] -> "0"
  | d :: dr ->
      string_of_int d :: List.map (fun d -> d |> string_of_int |> pad) dr |>
      String.concat ""

(* like :: on type int list but drops leading 0s *)
let zcons d ds =
  match d, ds with
  | 0, []-> []
  | _ -> d :: ds

let rec add ar br c =
  match ar, br, c with
  | ar, [], 0 -> ar
  | [], br, 0 -> br
  | [], [], c -> [c]
  | ar, [], c -> add ar [0] c
  | [], br, c -> add [0] br c
  | a :: ar', b :: br', c ->
      let d, c' =
        if a + b + c < base
        then a + b + c, 0
        else a + b + c - base, 1
      in
      d :: add ar' br' c'

let rec sub ar br c =
  match ar, br, c with
  | _, [], 0 -> ar
  | [], br, c -> raise (Invalid_argument "result is negative")
  | ar, [], c -> sub ar [0] c
```

```
      | a :: ar', b :: br', c ->
          let d, c' =
            if a - b - c >= 0
            then a - b - c, 0
            else a - b - c + base, 1
          in
          zcons d (sub ar' br' c')


    let (+) x y = add x y 0
    let (-) x y = sub x y 0

    let rec mul ar b =
      match ar, b with
      | _, 0
      | [], _ -> []
      | a :: ar, b ->
          from_int (a * b) + (0 :: mul ar b)

    let rec ( * ) x br = ???

  end
```

Try to understand this implementation. At a high level, we see that we represent natural numbers as entries in a list with base 10000. That is, if we have the list

$$[7890; 3456; 12]$$

then we are representing the number

$$7890 \times 10000^0 + 3456 \times 10000^1 + 12 \times 10000^2.$$

This representation allows us to store arbitrarily large numbers in a list such that the number can be recovered by doing

$$\sum_{i=0}^{arr.length-1} arr[i] \times 10000^i.$$

We also define the utility function `pad` which allows us to add any trailing zeroes when building the string of the represented number. Similarly, `zcons` allows us to drop preceeding zeroes when performing so that we do not obtain redundant representations such as

$$[7890; 3456; 12; 0]$$

and are always gauranteed to have a unique represntation.

For arithmetic we have the functions `add` and `sub` that perform addition and subtraction by iterating over both lists and performing any carries if necessary. In fact, these methods are equivalent to writing out the two numbers on a piece of paper and doing the addition via each unit place. However, we now work with every 10000-th number. Next, we see `(+)` and `(-)` which indicates to OCaml that these operations are infix operations.

Finally, we see the function `mul`. Can you deduce what it does? It is simply multiplying a big natural number by a constant `b` by traversing the list and multiplying each individual element and then summing the results. Your job is to complete the implementation of the function `( * )`. One possible implementation is given below.

```
let rec ( * ) x br =
  match x, br with
  | [], _
  | _, [] -> []
  | x, b :: br ->
      mul x b + (0 :: (x * br))
```

To multiply two big natural numbers we can simply traverse the list representation of one of them and multiply each element by the other big natural number. The result will be the sum of all such multiplications. Notice how this is similar to carrying out multiplication on pen-and-paper.

- Bonus question 1: Extend the `BigNat` module with any other operations you think would be useful. Some choices are division, modulo and the less than equal check. Make sure to update your signature too to let the client know these functions are available for use.

- We have implemented the module `BigNat` that allows us to get arbitrary precision arithmetic for natural numbers. We can use this module to implement another module that will implement arbitrary precision arithmetic for positive rational numbers. How can we go about this? A positive rational number consists of a numerator and a denominator where both are natural numbers and the denominator cannot be zero. This hints to us that a `BigPosRat` (standing for big positive rational) consists of a pair of `BigNat` where the first element of the pair is the numerator and the second element of the pair is the denominator. Now that we have an idea for what we would like for our module to look like, let's define the module's signature. The module must implement the same functions from `BigNat`. How should the signature look like? Take a go at it and then look at the answer which is given below.

```
(** Signature of module providing arbitrary precision arithmetic on positive rationals *)
module type BigPosRatType =
  sig
    type bigposrat

    (** The number 0 *)
    val zero: bigposrat

    (** The number 1 *)
    val one: bigposrat

     (** Convert an int to a big positive natural *)
    val from_int : int * int -> bigposrat

    (** Convert a big positive rational to an int pair (may overflow) *)
    val to_int : bigposrat -> int * int

    (** Convert a big positive rational to a string in decimal / decimal representation *)
    val to_string : bigposrat -> string

    (** Addition of big positive rationals *)
    val (+) : bigposrat -> bigposrat -> bigposrat

    (** Subtraction of big positive rationals. Raises [Invalid_argument] if result is negative. *)
    val (-) : bigposrat -> bigposrat -> bigposrat

    (** Multiplication of big positive rationals *)
    val ( * ) : bigposrat -> bigposrat -> bigposrat
  end
```

Note that we declared the type `bigposrat` rather than `(bignat,bignat)`. Why do we do this? The reason is that we do not want to expose our implementation to the client so that if we ever change the internal representation of the module `BigPosRat` the client code will still work.
Now we can implement the module `BigPosRat`. However, rather than implementing everything from scratch we can actually use the implementation of the module `BigNat` to reduce the amount of work we need to do. I will start you off with the starter code below.

```
module MakeBigPosRat(BigNat : BigNatType) : BigPosRatType =
  struct
    type bigposrat = BigNat.bignat * BigNat.bignat

    ...

  end
```

Try to complete the `...` with all the relevant functions. One possible solution is given below so you can compare your answers.

```
module MakeBigPosRat(BigNat : BigNatType) : BigPosRatType =
  struct
    type bigposrat = BigNat.bignat * BigNat.bignat

    (* Note that the denominator cannot be zero *)
    let zero = BigNat.zero, BigNat.one

    (* 1 / 1 *)
    let one = BigNat.one, BigNat.one

    let from_int (x,y) =
      if y = 0
      then raise (Invalid_argument "denominator cannot be zero")
      else BigNat.from_int x, BigNat.from_int y

    (* Caution: may overflow *)
    let to_int (n,d) = BigNat.to_int n, BigNat.to_int d

    let to_string (n,d) =
        BigNat.to_string n ^ " / " ^ BigNat.to_string d

    let (+) r1 r2 =
      match r1, r2 with
      | (n1, d1), (n2, d2) ->
          BigNat.(n1 + n2), BigNat.(d1 + d2)

    (** Will raise an exception if we get a negative value.
      * Can you see why?
      * It is because BigNat will raise an exception. *)
    let (-) r1 r2 =
      match r1, r2 with
      | (n1, d1), (n2, d2) ->
          BigNat.(n1 - n2), BigNat.(d1 - d2)
    let ( * ) r1 r2 =
      match r1, r2 with
      | (n1, d1), (n2, d2) ->
          BigNat.(n1 * n2), BigNat.(d1 * d2)

  end
```

That was easy wasn't it? Finally, we need to actually declare the module `BigPosRat` and this is straightforward.

```
module BigPosRat = MakeBigPosRat(BigNat)
```

At this point we are done.

- Note: There is one flaw in our implementation. Can you see what it is? For example, a subtraction may produce a number such as $(0, 452)$ or a number such as $(1231, 1231)$ which both can be either simplified or brought to a normal form ($(0, 452) = (0, 1)$). Furthermore, if we wish to build an even better module then numbers such as $(6, 2)$ should be simplified to $(3, 1)$. You can try implementing these improvements as a challenge.

- You can also find the file `test.ml` to try out the code we just implemented. You can either run the REPL and input the following commands

  ```
  #use "arithmetic.ml"
  #use "test.ml"
  ```

  or actually compile and run the program with the following commands

  ```
  ocamlopt -o test arithmetic.ml test.ml
  ./test
  ```

  on a terminal window. Note that you can also compile using `ocamlc` which will give you platform independent bytecode rather than native code which is what `ocamlopt`. This looks as follows.

  ```
  ocamlc -o test arithmetic.ml test.ml
  ./test
  ```

- Bonus question 2: Extend your implementation of `BigPosRat` using your extended implementation of `BigNat`.

- Bonus question 3: Try implementing a new module `BigInt` so as to get big integer numbers. That is, this module will allow for arbitrary precision arithmetic of positive and negative numbers. You have two choices here, you can either build from scratch or implement a functor that uses `BigNat`. Hint: `type bigint` can be a pair of type `bool * BigNat.bignat` where the first element of the pair indicates to us whether the number is positive or negative.

- Bonus question 4: Try implementing a new module for arbitrary precision rational numbers (not just positive ones). What's the best way to go about this? Consider using your implementation of `BigInt` from question 3 and then creating functor to make a `BigRat`.