Homework 4

Functional Programming (ITI0212)

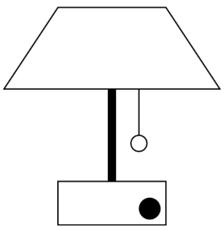
due: 2022-05-04

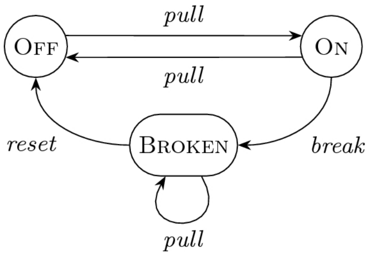
Place your solutions in a module named `Homework4` in a file with path `homework/Homework4.idr` within your `iti0212-2022` repository on the TalTech GitLab server. Your solutions will be pulled automatically for marking shortly after the due date.

At the start of the file include a comment containing your name as it appears in your university records. Precede each problem's solution with a comment specifying the problem number.

The solution file that you submit should load without errors. If you encounter a syntax or type error that you are unable to resolve, please use comments or holes to isolate them from the part of the file interpreted by Idris.

Problem 1
Recall the finite state transition system for a lamp from homework 3, problem 6.





We can represent this as a dependent type indexed by the type of its possible states:

```
data  State : Type  where
  On  :  State
  Off  :  State
  Broken  :  State

data  Lamp : State -> Type  where
  OnLamp  :  Lamp On
  OffLamp  :  Lamp Off
  BrokenLamp  :  Lamp Broken
```

Copy these definitions into your code file to complete the following tasks.

- Write the functions represented by the state transitions `break` and `reset`:

  ```
  break  :  Lamp On -> Lamp Broken

  reset  :  Lamp Broken -> Lamp Off
  ```

- Write a function that computes the effect of pulling the string on the lamp's state:

  ```
  pulled  :  State -> State
  ```

- Write the function represented by the state transition `pull`:

```
pull  :  Lamp state -> Lamp (pulled state)
```

Problem 2
In this problem we will consider a different way of representing `Either` types. You should `import Data.Fin` and copy into your file the following definition, which returns either its first or its second argument, depending on which `Fin 2` appears as the third argument:

```
case2  :  a -> a -> Fin 2 -> a
case2 x y 0  =  x
case2 x y 1  =  y
```

The following type constructor is in fact isomorphic to `Either`:

```
Either2  :  Type -> Type -> Type
Either2 a b  =  DPair (Fin 2) (\ i => case2 a b i)
```

To verify this, write the following functions:

```
from_Either  :  Either a b -> Either2 a b

to_Either  :  Either2 a b -> Either a b
```

so that their composition, in either order, acts as the identity function.
For example:

```
Homework4> (to_Either . from_Either) (Left 42)
Left 42
Homework4> (to_Either . from_Either) (Right "hello")
Right "hello"
Homework4> (from_Either . to_Either) (0 ** 42)
MkDPair FZ 42
Homework4> (from_Either . to_Either) (1 ** "hello")
MkDPair (FS FZ) "hello"
```

Interacting with a simple database

In the following problems we will interact with a simple user database of some social networking platform. The "database" stores `User`s...

```
UserId : Type
UserId = Bits64

Feed : Type
Feed = Visibility (List Post)

record User where
  constructor MkUser
  id : UserId
  name : String
  feed : Feed
```

...who each have their own `Feed`, which can either be private or contain a list of `Post`s:

```
record Votes where
  constructor MkVotes
  likes : Bits64
  dislikes : Bits64

record Post where
  constructor MkPost
  votes : Votes
  timestamp : Bits64
  -- ... potentially more fields

data Visibility : (a : Type) -> Type where
  Private : Visibility a
  Public : (content : a) -> Visibility a
```

Each post has `Votes`, which you might recognize from Lab 9. *Note: `Bits64` is a type of 64-bit unsigned integers. It's used instead of `Nat` since it computes much quicker.*

Problem 3
Begin by writing some code that summarizes user data. You have three tasks:

1. Give an implementation of `Monoid Votes` that combines two `Votes` by summing up their likes and dislikes.
2. Implement `Functor Visibility` to apply functions to values that are `Public`.
3. Combine the two to calculate a user's total score, i.e. the difference between all likes and dislikes they received if their posts were public:

   ```
   total_score : User -> Visibility Integer
   ```

Test this with a few example users.

```
alice : User
alice = MkUser 1 "Alice" Private

bob : User
bob = MkUser 2 "Bob" $ Public []

eve : User
eve = MkUser 42 "Eve" $ Public
  [ MkPost (MkVotes 30 9) 1643580000
  , MkPost (MkVotes 321 27) 1650282774
  ]
```

Alice's posts are private, *Bob* hasn't posted anything yet and *Eve* has posted twice.

```
Homework4> total_score alice
Private
Homework4> total_score bob
Public 0
Homework4> total_score eve
Public 315
```

Hint: You can adapt the functions `score` (Lab 9) and `fold_list` (Lab 4) to calculate a user's score.

Problem 4
In this problem, we introduce a simple form of error handling to ease interaction

with a user database. Your task will be to copy some code, and to make it run by implementing `Monad` for a given type.

Consider the following type, representing a result value obtained from the database:

```
data DbError : Type where
  Unauthorized : DbError
  NotFound : DbError

data DbResult : (a : Type) -> Type where
  Err : (err : DbError) -> (msg : String) -> DbResult a
  Ok : a -> DbResult a
```

A `DbResult` is either an error that contains an error code and an error message, or the result of a successful operation, for example:

```
missing_user : DbResult a
missing_user = Err NotFound "User 'Claire' does not exist"

bobs_friends : DbResult (List String)
bobs_friends = Ok ["Alice", "Eve", "Michael"]
```

We use this type to model database accesses that might fail. We define the following (full definitions at the end of the problem, simply copy them from there):

- The "database", for simplicity this is just a list of users:

  ```
  user_database : List User
  ```

- A function that, given a user ID, returns `Ok user` if a user with that ID was found in the database, or `Err NotFound "..."` if not:

  ```
  get_user : UserId -> DbResult User
  ```

- A function that retrieves a user's posts if they are public, otherwise returns an error:

  ```
  get_posts : User -> DbResult (List Post)
  ```

- A function that composes the above and retrieves a user's most recent post:

  ```
  get_latest_post : UserId -> DbResult Post
  ```

Your task:

1. Copy the definitions of `DbError`, `DbResult`, `user_database`, `get_user`, `get_posts` and `get_latest_post` from below. You need to import `find` from `Data.List`, too.

2. Implement `Functor`, `Applicative` and `Monad` for the type family `DbResult`.

   Your implementation should allow us to write the *happy path* of a function in `do`-notation. Instead of having to explicitly pattern match on `DbResult`s like so...

   ```
   -- Do `case ... of` a bunch of times and
   -- things will become unbearably repetetive.
   bad : UserId -> DbResult (List Post)
   bad user_id = case get_user user_id of
                      (Err err msg) => Err err msg
                      (Ok user) => get_posts user
   ```

...we want to give definitions that bind success values (wrapped in `Ok`) and return the first `Err` they encounter.

The function `get_latest_post` below is written in the desired *happy path* style.

The code: Please copy this, together with the definitions of `DbError` and `DbResult`:

```
user_database : List User
user_database = [ alice , bob , eve ]

get_user : UserId -> DbResult User
get_user id =
  case find (\u => u .id == id) user_database of
       Nothing => Err NotFound $
                    "No user with ID "
                      ++ show id
                      ++ " exists"
       Just user => Ok user

get_posts : User -> DbResult (List Post)
get_posts user =
  case user .feed of
       Private => Err Unauthorized $
                    "Posts of user'"
                      ++ user .name
                      ++ "' are set to private"
       Public posts => Ok posts

-- Using `do` notation requires `Monad DbResult`
get_latest_post : UserId -> DbResult Post
get_latest_post user_id = do

  -- The "happy path":
  -- The bound value `user` is of type `User`,
  -- `posts` has type `List Post`, any
  -- error is implictly returned.
  user <- get_user user_id
  posts <- get_posts user

  -- This is never evaluated if any of the preceeding
  -- functions return an error:
  foldl
    (Ok .: compare_newer)
    (Err NotFound "No posts for user \{user .name}")
    posts
  where
    compare_newer : DbResult Post -> Post -> Post
    compare_newer (Err _ _) post = post
    compare_newer (Ok post) post' =
      if post .timestamp > post' .timestamp
         then post
         else post'
```

You should not worry about any new concepts (`(.:)`, `foldl`, string interpolation `"\{...}"`), but are encouraged to look up their documentation.

Examples: Retrieving posts for a user that does not exist should return an error:

```
Homework4> get_latest_post 213
Err NotFound "No user with ID 213 exists"
```

Similarly, for user that exists but have their posts set to private:

```
Homework4> get_latest_post 1
Err Unauthorized "Posts of user 'Alice' are set to
   private"
```

If the user exists, and has multiple posts, the post with the most recent timestamp is returned:

```
Homework4> get_latest_post 42
Ok (MkPost (MkVotes 321 27) 1650282774)
```

Properties of alternating lists

Problem 5
In this problem, you will prove properties of *alternating lists* by encoding the proposition "a list is an alternation of values x and y" as an indexed type.

You are *not* required to come up with an overly clever definition. Instead, the goal of this problem is that you demonstrate your ability to encode a *proposition as a type*, and to use its *proofs like ordinary terms*. Both the definition of this proposition and the statements that it entails should be very similar to examples you have already seen (either in a lecture or the labs); please feel free to draw as much inspiration from those as necessary.

Definition: Given terms `x, y : a`, we define *alternating lists of* `x` *and* `y` inductively:

1. The empty list `[] : List a` is an alternating list of `x` and `y`.
2. If `zs : List a` is an alternating list of `x` and `y`, then `(x :: y :: zs)` is, too.

You have the following tasks:

1. Define an indexed type

   ```
   data IsAlternating :
     (x , y : a) -> (zs : List a) -> Type where
     -- Your constructors go here...
   ```

   that encodes the above definition. *Note: The syntax `(x , y : a)-> ...` is shorthand for `(x : a)-> (y : a)-> ....` Use it if you have many repeated arguments of the same type.*

2. Prove that the following example list is indeed alternating:

   ```
   ex_alt : IsAlternating 1 0 [1, 0, 1, 0]
   ```

3. Write a function that counts how many alternations of `x` and `y` an alternating list contains:

   ```
   count_alternations : (zs : List a)
     -> (is_alt : IsAlternating x y zs)
     -> Nat
   ```

   The function should evaluate to 2 for the example given in task 2:

   ```
   Homework4> count_alternations [1, 0, 1, 0] ex_alt
   2
   ```

If you want to test it with a different list, you'll have to come up with a proof `?prf` yourself:

```
Homework4> count_alternations [2, 4, 2, 4, 2, 4] ?prf
3
```

4. Prove that, assuming `zs` and `ws` are alternating, appending one to the other results in an alternating list:

```
isAlternatingAppend : {x , y : a}
  -> (zs , ws : List a)
  -> (alt_zs : IsAlternating x y zs)
  -> (alt_ws : IsAlternating x y ws)
  -> IsAlternating x y (zs ++ ws)
```

Hint: Try induction on the proof `alt_zs`.