Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 6

In this exercise we are going to look at functors and monads. This exercise sheet is a bit longer, but important as many subsequent chapters of this lecture build on monads.

Functors

In Haskell, a *functor* is represented by a type constructor `f` of kind `* -> *`, i.e. something that takes a type and returns a type, e.g. `List` or `Maybe`, together with an operation

```
fmap :: (a -> b) -> (f a -> f b)
```

which satisfies the functor laws:

```
fmap g . fmap h  =  fmap (g . h)
fmap id          =  id
```

Usually, a good intuition for a functor is something, which behaves like a container, i.e. that `f a` describes some kind of container with elements of type `a`. With this intuition, `fmap g xs` means applying the function `g` to each element of the container. The functor laws ensure that this is all that `fmap` does, e.g. that `fmap` for lists does not change the container structure, e.g. by removing or duplicating elements of the list.

Monads

In Haskell, a *monad* is represented by a functor `m` together with two operations

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

which satisfy the monad laws:

```
mx >>= return      =  mx
return x >>= f     =  f x
(mx >>= g) >>= h   =  m >>= (\x -> g x >>= h)
```

Usually, a good intuition for a value of type `m a` is something, which behaves like an effectful computation that when run produces a result of type `a`. Which effects these computations can cause depends on the monad `m` itself, e.g. `Maybe` models computations, which can fail, and `State` models computations, which can implicitly read and write from some mutable state.

Every monad is also a functor, but not vice-versa: by stretching our intuition of a container, we can view a computation that produces a value of type `a` as a container with elements of type `a`. Mapping a function over such a computation, will yield another computation, which first runs the original computation, potentially causing effects, and then applying the function to the result without causing additional effects.

Type Class Hierarchy

Functors and monads form a type class hierarchy, similar as we have seen with semigroups and monoids in a previous exercise. However, in the standard library, there is another concept in-between, called an *applicative functor*. Each monad is an applicative functor, and each applicative functor is a functor, but not vice versa. Hence, the type class hierarchy looks as followed:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

As we did not cover applicative functors yet, you can for now just leave the `pure` and `(<*>)` functions `undefined` when writing instances for monads, e.g.

```
instance Functor Maybe where
  fmap = myFmap

instance Applicative Maybe where
  pure = undefined
  (<*>) = undefined

instance Monad Maybe where
  return = myReturn
  (>>=) = myBind
```

Note, that you might get a warning when implementing `return`. You can ignore this warning, as we will get back to that when we look at applicative functors.

Kleisli Categories

To better understand the monad laws and strengthen our intuition of monads, it can be useful to look at an alternative definition of monads. One such definition is that of a *Kleisli Category*, where we keep `return`, but replace the bind operator

```
(>>=) :: m a -> (a -> m b) -> m b
```

with a composition operator

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Contrast the type of the `(<=<)` operator with that of regular function composition:

```
(.)   :: (b ->   c) -> (a ->   b) -> (a ->   c)
```

The `(<=<)` operator allows us to compose functions with side effects of type `a -> m b`, just like they were regular functions of type `a -> b`. Compared to the regular function composition, the `(<=<)` operator not only feeds the output of the second function to the input of the first function, but also composes the effects of both functions for us.

This allows us for each monad `m` to form a category, where

- objects are Haskell types;
- arrows `a -> b` are Haskell functions of type `a -> m b`;
- for each type `a`, the identity arrow is `return :: a -> m a`; and
- the composition operator is `(<=<)`.

This category is called the *Kleisli category for* `m` and its category laws are equivalent to the monad laws, but easier to understand:

```
  return <=< f      =  f
  f <=< return      =  f
  (f <=< g) <=< h  =  f <=< (g <=< h)
```

Sticking with our intuition of monads as effectful computations, the first two laws ensure that `return` is not allowed to introduce effects, and the third law ensures that parentheses are not allowed to influence how effects are composed.

As our original definition of monads and the one as Kleisli categories are equivalent, we can derive the `(<=<)` operator from `(>>=)`

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```

and vice versa

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
mx >>= f = (f <=< (\_ -> mx)) ()
```

Monad APIs

In Haskell, monads are usually defined in their own module exposing an API following a certain pattern. The following illustrates this by showing this pattern for the `State` monad:

```
newtype State s a = ...

instance Functor (State s) where ...
instance Applicative (State s) where ...
instance Monad (State s) where ...

runState :: State s a -> s -> (a, s)

get    :: State s s
put    :: s -> State s ()
modify :: (s -> s) -> State s ()
```

The basic idea is that a user of this monad never constructs a `State` value by hand, but instead uses the `get`, `put`, and `modify` functions to construct primitive computations. Those computations can then be manipulated and combined by using the `Functor` and `Monad` instances, and are finally executed via the `runState` function.

In this case, the primitive computations are:

- `get`, which retrieves the current state, i.e. it is a computation which keeps a state of type `s` and returns a value of type `s`;
- `put`, which assigns a new value to the current state, i.e. it is function which takes a new state of type `s`, and returns a computation which keeps a state of type `s` and returns the uninteresting value `()`; and
- `modify`, which takes a function on states and returns a state computation, which changes the current state by applying the function to it.

To run a state computation via `runState`, we need to provide an initial state value, and in return get the result of the computation and the final state value.

This API pattern is followed for most monads, with the notable exceptions of `Maybe` and sometimes `Either`.

We will follow this API pattern in the whole exercise, as it makes it clear when we are thinking of a value of type `m a` as just a value of type `m a` vs a computation that produces values of type `a` when run.

Exercise 1 (Monad Instances & Applications)

In this exercise, your task is to implement various monads by following the beforementioned API pattern, and then use their API to write simple functions in monadic style.

1. Lists form a monad that represents non-deterministic (`ND`) computations. The API for the `ND` monad is as followed:

   ```
   newtype ND a = ND [a]

   instance Functor ND where ...
   instance Applicative ND where ...
   instance Monad ND where ...

   runND :: ND a -> [a]

   choose :: [a] -> ND a
   abort  :: ND a
   ```

   The `choose` function takes a list and returns a `ND` computation, which non-deterministically chooses an element from the list.

   The `abort` function returns a `ND` computation, which will invalidate non-determinstic choices of previous `ND` computations, by choosing an element of the empty list.

   Running a `ND` computation yields the list of all possible results that the non-determinstic computation could produce. This is possible because each `ND` computation *is* the list of its possible outcomes.

   Note that `runND`, `choose`, and `abort` are trivial to implement, but the difficulty is in understanding how lists represent non-determinstic computations.

   Examples:

   ```
   ex1 :: [Int]
   ex1 = runND $ do
     x <- choose [1, 2]
     y <- choose [10, 20]
     return $ x + y

   -- ex1 == [11, 21, 12, 22]

   ex2 :: [Int]
   ex2 = do runND $ do
     x <- choose [1..10]
     if even x then
       return x
     else
       abort

   -- ex2 == [2, 4, 6, 8, 10]
   ```

Your task is to implement the ND monad as described above and then use it to

- write a function

  ```
  flipCoin :: ND Bool
  ```

  which flips a coin by non-deterministically choosing a boolean.
- write a function

  ```
  flipTwoCoins :: ND (Bool, Bool)
  ```

  which flips two coins by using the `flipCoin` function.
- Write a function, which solves the graph coloring problem.

  We represent a graph as a map[1], which maps each node to the list of its neighbors.

  ```
  type Graph n = [(n, [n])]
  ```

  We represent a coloring as a map from nodes to colors

  ```
  type Coloring n c = [(n, c)]
  ```

  Given a graph and a list of colors, solving the graph coloring problem means assigning each node a color, such that all neighbors of that node have different colors.

  This problem can be solved by using backtracking (`choose` and `abort`) in 9 lines of Haskell code.

  Hint: you might want to use a helper function as followed and recursively walk through the graph:

  ```
  solve :: (Eq n, Eq c) => Graph n -> [c] -> ND (Coloring n c)
  solve g colors = solve' g colors [] where
    solve' g colors coloring = ...
  ```

  Example:

  ```
  exGraph :: Graph Int
  exGraph =          --    1
    [ (0, [1,2])     --   / \
    , (1, [3,0])     --  0   3
    , (2, [3,0])     --   \ /
    , (3, [1,2])     --    2
    ]

  exColorings :: [Coloring Int String]
  exColorings = runND $ solve exGraph ["red", "blue"]

  -- exColorings is
  -- [ [(3,"red"),  (2,"blue"), (1,"blue"), (0,"red")]
  -- , [(3,"blue"), (2,"red"),  (1,"red"),  (0,"blue")]
  ```

[1] To avoid using the `containers` library, we simply represent a map as an association list, i.e. a list of key-value-pairs. The `lookup` function allows to retrieve a value from a key and is imported by default:

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
```

```
    -- ]
```

2. The `Maybe` type forms a monad that represents partiality, i.e. computations which may fail. A total function of type `a -> Maybe b` can be seen as a partial function of type `a -> b`.

   The API for the partiality monad is as followed:

   ```
   newtype Partial a = Partial (Maybe a)

   runPartial :: Partial a -> Maybe a

   instance Functor Partial where ...
   instance Applicative Partial where ...
   instance Monad Partial where ...

   failure :: Partial a
   ```

   Your task is to implement the `Partial` monad as described above and then use it to

   - write a function

     ```
     (!?) :: [a] -> Int -> Partial a
     ```

     such that `xs !? i` tries to retrieve the element at index `i` of the list `xs` and fails if the index is out of bounds.

   - write a function

     ```
     getCell :: [[a]] -> Int -> Int -> Partial a
     ```

     which takes a matrix (list of rows) and a `x` and `y` coordinate and tries to retrieve the cell at row `y` and column `x`. Use `do` notation and the `(!?)` operator from the previous sub-exercise.

3. The `Either e` type forms a monad that represents computations which may fail with an exception of type `e`.

   Recall, that the `Either` type is defined as

   ```
   data Either a b = Left a | Right b
   ```

   The API for the exception monad is as followed:

   ```
   newtype Exception e a = Exception (Either e a)

   runException :: Exception e a -> Either e a

   instance Functor Partial where ...
   instance Applicative Partial where ...
   instance Monad Partial where ...

   raise         :: e -> Exception e a
   withException :: Partial a -> e -> Exception e a
   ```

   The `withException` function allows to convert between the `Partial` and `Exception e` monad by providing an exception value for the failure case of the partiality monad.

   Your task is to implement the `Exception e` monad as described above and then use it to

- write a function

  ```
  validatePassword :: String -> Exception String ()
  ```

  which takes a password and checks if it is at least 8 characters long and if it contains both letters and digits. If one of those conditions is not satisfied, it should raise an exception consisting of a string, which describes the reason for failure.

- rewrite the `getCell` function from the previous subexercise such that if `getCell` fails, it will signal whether the row or the column index was out of bounds, e.g.

  ```
  data MatrixError = InvalidRowIndex | InvalidColIndex
  getCell' :: [[a]] -> Int -> Int -> Exception MatrixError a
  ```

  Use the `withException` function in combination with the `(!?)` function from the previous exercise.

4. Functions of type `s -> (a, s)` form a monad in `a` that represents computations, which are allowed to manipulate an implicit state of type `s`.

   The API for the state monad is as followed:

   ```
   newtype State s a = State (s -> (a, s))

   instance Functor (State s) where ...
   instance Applicative (State s) where ...
   instance Monad (State s) where ...

   runState :: State s a -> s -> (a, s)

   get    :: State s s
   put    :: s -> State s ()
   modify :: (s -> s) -> State s ()
   ```

   Note that the `State` monad merely encapsulates how a function in a pure functional language allows to work with state in the first place, e.g. a function which takes an argument of type `Int`, modifies a `String`, and returns a result of type `Bool` is something, which we would naturally express in a functional language as

   ```
     f :: Int -> String -> (Bool, String)
   ```

   The state monad just encapsulates this pattern of threading the `String` data through the function and allows us to write

   ```
     f :: Int -> State String Bool
   ```

   and then use `do`-notation inside of `f` to access and modify the `String` value in an imperative style.

   Your task is to implement the `State s` monad as described above and then implement the following:

   - On the website you can find the file `WhileInterp.hs`, which contains an interpreter for a simple language with variables, while-loops, and assignments.

     The datatypes are as followed:

```haskell
type Var = String

data Val = VInt Int
         | VBool Bool
         | VUnit
         | VError String
         deriving (Eq, Show)

data Op = Add | Sub | Less
        deriving (Eq, Show)

data Expr = EVar Var
          | EVal Val
          | EOp Expr Op Expr
          | EAssign Var Expr
          | EWhile Expr Expr
          | ESeq Expr Expr
          deriving (Eq, Show)

type Env = [(Var, Val)]
```

Variables are represented as strings. Values are integers, booleans, unit (like `()` in Haskell) and error values carrying an error message. Binary operators are addition, subtraction, and inequality. An expression is either a variable `EVar x`, a value `EVal v`, an application of a binary operator `EOp e1 op e2`, an assignment expression `EAssign x e`, a while-expression `EWhile e1 e2`, or a sequence expression `ESeq e1 e2`. Environments map each variable to its current value, and are used during evaluation to propagate variable values from assignments to variable uses.

To keep it simple, we do not distinguish between statements and expressions, but instead assignment and while-loops are simply expressions, which evaluate to the unit value. The `ESeq e1 e2` expression acts like a semicolon, and will first evaluate `e1`, then throw the resulting value away and evaluate `e2`.

If a binary operation is called with arguments of incorrect types, e.g. adding an integer and a boolean, it will evaluate to a `VError` value. Similarly, if a `VError` value appears as the argument of an operator application, the whole operator application will evaluate to that `VError` value, which propagtes the error further to the root of the expression tree.

The following shows a program in this language first in pseudo-code, and then as an expression of type `Expr`:

```
Pseudo-Code:
  x = 0;
  while x < 10 (x = x + 1);
  x + 5


Haskell:
  example :: Expr
  example = ESeq (EAssign "x" $ EVal $ VInt 0)
                 (ESeq (EWhile (EOp (EVar "x") Less (EVal $ VInt 10))
                               (EAssign "x" $ EOp (EVar "x") Add (EVal $ VInt 1)))
                       (EOp (EVar "x") Add (EVal $ VIn 5)))
```

Evaluating this expression in the empty environment yields the value of the expression and the final environment, i.e. the values of the variables after the evaluation finished:

```
>>> eval example []
(VInt 15, [("x", VInt 10)])
```

Rewrite the `eval` function from `WhileInterp.hs` in monadic style by using the `State Env` monad to avoid threading the `Env` through the function by hand:

```
eval' :: Expr -> State Env Val
eval' = ...
```