AFP Assignments

(version: October 21, 2013)

Atze Dijkstra

(and Andres Löh, Doaitse Swierstra, ...)

Sep-Oct 2013

This is the set of assignments for the AFP course. From it are taken subsets which have to be made and submitted throughout the AFP course. The exercises are roughly categorized and partitioned by their main topic, but there is overlap in this categorization. Also, there is no ordering in the complexity or difficulty of each exercise. On the AFP website you can find which of these exercises have to be made when, and how submission is done.

Contents

1 Tooling

Exercise 1.1 (Cabal)**.** Submit your solutions to the assignments as a Cabal package. Turn the code (for the `perms_smooth` task, see exercise 7.1) into a library. Include the solutions to the theoretical packages as extra documentation files. Write a suitable package description, and include a Setup script so that the package can easily be built and installed using Cabal. As a package name, choose a name that includes your CS logins. Produce the file using the command `cabal sdist`.

2 Programming

Exercise 2.1 ((Tail) recursion)**.** [15%] Consider the datatype

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

The function `splitleft` splits off the leftmost entry of the tree and returns that entry as well as the remaining tree:

```
splitleft :: Tree a -> (a, Maybe (Tree a))
splitleft (Leaf a)   = (a, Nothing)
splitleft (Node l r) = case splitleft l of
                         (a, Nothing) -> (a, Just r)
                         (a, Just l') -> (a, Just (Node l' r))
```

Write a *tail-recursive* variant of `splitleft`.

Hint. Generalize `splitleft` by introducing an additional auxiliary parameter. Recall that functions can be used as parameters. If you do not know what "tail-recursive" means, look up the definition somewhere.

Exercise 2.2 (Tree unfold)**.** Recall `unfoldr`:

```
unfoldr :: (s -> Maybe (a, s)) -> s -> [a]
unfoldr next x =
  case next x of
    Nothing    -> []
    Just (y, r) -> y:unfoldr next r
```

We can define an unfold function for trees as well:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
  deriving Show

unfoldTree :: (s -> Either a (s, s)) -> s -> Tree a
unfoldTree next x =
  case next x of
    Left y      -> Leaf y
    Right (l, r) -> Node (unfoldTree next l) (unfoldTree next r)
```

Task. Define the following functions using `unfoldr` or `unfoldTree`:

```
iterate :: (a -> a) -> a -> [a]
```

(The call `iterate f x` generates the infinite list `[x, f x, f (f x), ...]`.)

```
map :: (a -> b) -> [a] -> [b]
```

(As defined in the prelude.)

```
balanced :: Int -> Tree ()
```

(Generates a balanced tree of the given height.)

```
sized :: Int -> Tree Int
```

(Generates any tree with the given number of nodes. Each leaf should have a unique label.)

Exercise 2.3 (Use of fix)**.** Given the function

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

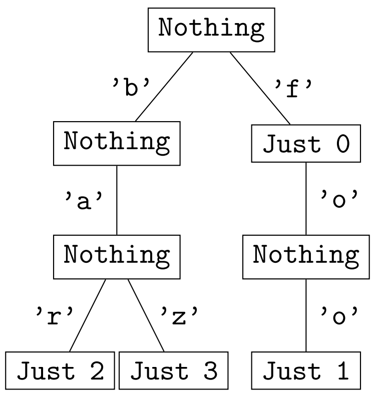
Define the function `foldr` as an application of `fix` to a term that is not recursive.

Exercise 2.4 (Trie)**.** A *trie* (sometimes called a prefix tree) is an implementation of a finite map for structured key types where common prefixes of the keys are grouped together. Most frequently, tries are used with strings as key type. In Haskell, such a trie can be represented as follows:

```
data Trie a = Node (Maybe a) (Map Char (Trie a))
```

A node in the trie contains an optional value of type `a`. If the empty string is in the domain of the trie, the associated value can be stored here. Furthermore, the trie maps single characters to subtries. If a key starts with one of the chracters contained in the map, then the rest of the key is looked up in the corresponding subtrie.

The following picture shows an example trie that maps `"f"` to `0`, `"foo"` to `1`, `"bar"` to `2` and `"baz"` to `3`:



The implementation should obey the following invariant: if a trie (or subtrie) is empty, i.e., if it contains no values, it should always be represented by

```
Node Nothing Data.Map.empty
```

Task. Write a module `Data.Trie` containing a datatype of tries as above and the following functions:

```
empty  :: Trie a
null   :: Trie a -> Bool
valid  :: Trie a -> Bool
insert :: String -> a -> Trie a -> Trie a
lookup :: String -> Trie a -> Maybe a
delete :: String -> Trie a -> Trie a
```

The function `valid` tests if the invariant holds. All operations should maintain the invariant.

Exercise 2.5 (Type hiding)**.** Define a function `count` such that the following program is well-typed

```
test :: [Int]
test = [count, count 1 2 3, count "" [True, False] id (+)]
```

and evaluates to `[0, 0, 0]`. In other words, `count` should accept an arbitrary number of arguments (of arbitrary types), and just always return `0`.

Then redefine the function `count` such that `test` evaluates to `[0, 3, 4]`, i.e., `count` should return the number of arguments.

Please submit both versions of the function.

Hint: no language extensions are required to solve this exercise.

Exercise 2.6 (Partial computation)**.** The type constructor `Partial` can be used to describe possibly nonterminating computations in such a way that they remain productive.

```
data Partial a = Now a | Later (Partial a)
  deriving Show
```

We can now describe a productive infinite loop as follows:

```
loop = Later loop
```

Even functions that terminate can produce a `Later` for every step they perform, allowing us to judge the complexity of the computation afterwards by looking at how many `Later` occurrences there are before the result.

```
runPartial :: Int -> Partial a -> Maybe a
runPartial _ (Now x)   = Just x
runPartial 0 (Later p) = Nothing
runPartial n (Later p) = runPartial (n - 1) p
```

Using `runPartial`, we can then run a partial computation and give a bound on the number of steps it is allowed to perform. If no result is delivered in the allowed number of steps, Nothing is returned. Note that runPartial n loop terminates for all non-negative choices of n.

There also is

```
unsafeRunPartial :: Partial a -> a
unsafeRunPartial (Now x)   = x
unsafeRunPartial (Later p) = unsafeRunPartial p
```

that runs a partial computation with the risk of nontermination.

The Partial type constructor forms a monad:

```
instance Monad Partial where
  return      = Now
  Now x   >>= f = f x
  Later p >>= f = Later (p >>= f)
```

The function tick introduces an explicit delay:

```
tick = Later (Now ())
```

Here is another example:

```
psum :: [Int] -> Partial Int
psum xs = liftM sum (mapM (\x -> tick >> return x) xs)
```

Try psum on a couple of lists to see what it does.

It also forms a MonadPlus:

```
class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Here, mplus gives us a form of addition (choice) between two partial computations. The idea is that both computations are run in parallel, and the one terminating earlier is returned. The neutral element of this form of choice is loop, the function that never returns.

```
instance MonadPlus Partial where
  mzero = loop
  mplus = merge

merge :: Partial a -> Partial a -> Partial a
merge (Now x)   _         = Now x
merge _         (Now x)   = Now x
merge (Later p) (Later q) = Later (merge p q)
```

Use this monad as well as mzero and mplus (or derived functions) to define a function

```
firstsum :: [[Int]] -> Partial Int
```

that performs psum on every of the integer lists and returns the result that can be obtained with as few delays as possible.

Example:

```
runPartial 100 $ firstsum [repeat 1, [1, 2, 3], [4, 5], [6, 7, 8], cycle [5, 6]]
```

returns Just 9.

Unfortunately, firstsum will not work on infinite (outer) lists and

```
runPartial 200 $ firstsum (cycle [repeat 1, [1, 2, 3], [4, 5], [6, 7, 8], cycle [5, 6]])
```

will loop. The problem is that merge schedules each of the alternatives in a fair way. When using merge on an infinite list, all computations are evaluated one step before the first Later is produced. The solution is to write an unfair merge. Rewrite merge such that the above test returns Just 9 and also

```
runPartial 200 $ firstsum (replicate 100 (repeat 1) ++ [[1]] ++ repeat (repeat 1))
```

returns Just 1.

Exercise 2.7 (Object-orientation). Using open recursion and an explicit fixed-point operator similar to

```
fix f = f (fix f)
```

we can simulate some features commonly found in OO languages in Haskell. In many OO languages, objects can refer their own methods using the identifier this, and to methods from a base object using super.

We model this by abstracting from both this and super:

```
type Object a = a -> a -> a
data X = X {n :: Int, f :: Int -> Int}
x, y, z :: Object X
x super this = X {n = 0, f = \i -> i + n this}
y super this = super {n = 1}
z super this = super {f = f super . f super}
```

We can extend one "object" by another using extendedBy:

```
extendedBy :: Object a -> Object a -> Object a
extendedBy o1 o2 super this = o2 (o1 super this) this
```

By extending an object o_1 with an object o_2, the object o_1 becomes the super object for o_2.

Once we have built an object from suitable components, we can close it to make it suitable for use using a variant of `fix`:

```
fixObject o = o (error "super") (fixObject o)
```

We close the object `o` by instantiating it with an error super object and with itself as `this`.

Look at what the type of `fixObject` is and familiarize yourself with the behaviour of `fixObject` by trying the following expressions:

```
n (fixObject x)
f (fixObject x) 5
n (fixObject y)
f (fixObject y) 5
n (fixObject (x `extendedBy` y))
f (fixObject (x `extendedBy` y)) 5
f (fixObject (x `extendedBy` y `extendedBy` z)) 5
f (fixObject (x `extendedBy` y `extendedBy` z `extendedBy` z)) 5
```

Task. Define an object

```
zero :: Object a
```

such that for all types `t` and objects `x :: Object t`, the equation `x `extendedBy` zero` $\equiv$ `zero `extendedBy` x` $\equiv$ `x` hold. (You do not need to provide the proof.)

A more interesting use for these functional objects is for adding effects to functional programs in an aspect-oriented way.

In order to keep a function extensible, we write it as an object, and keep the result value monadic:

```
fac :: Monad m => Object (Int -> m Int)
fac super this n =
  case n of
    0 -> return 1
    n -> liftM (n*) (this (n - 1))
```

Note that recursive calls have been replaced by calls to `this`. We can now write a separate aspect that counts the number of recursive calls:

```
calls :: MonadState Int m => Object (a -> m b)
calls super this n =
  do
    modify (+1)
    super n
```

We can now run the factorial function in different ways:

```
runIdentity (fixObject fac 5)                      ≡ 120
runState    (fixObject (fac‘extendedBy‘ calls) 5) 0 ≡ (120, 6)
```

Task. Write an aspect `trace` that makes use of a writer monad to record whenever a recursive call is entered and whenever it returns. Also give a type signature with the most general type. Use a list of type

```
data Step a b = Enter a
              | Return b
  deriving Show
```

to record the log. As an example, the call

```
runWriter (fixObject (fac‘extendedBy‘ trace) 3)
```

yields

```
(6, [Enter 3, Enter 2, Enter 1, Enter 0, Return 1, Return 1, Return 2, Return 6])
```

Exercise 2.8 (Enumeration)**.** Consider the following datatype:

```
data GP a = End a
          | Get (Int -> GP a)
          | Put Int (GP a)
```

A value of type `GP` can be used to describe programs that read and write integer values and return a final result of type `a`. Such a program can end immediately (`End`). If it reads an integer, the rest of the program is described as a function depending on this integer (`Get`). If the program writes an integer (`Put`), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

```
echo = Get (\n -> Put n echo)
```

Task. Write a function

```
run :: GP a -> IO a
```

that can run a `GP`-program in the `IO` monad. A `Get` should read an integer from the console, and `Put` should write an integer to the console.

Here is an example run from GHCi:

```
>>> run echo
? 42
```

```
42
? 28
28
? 1
1
? -5
-5
? Interrupted.
>>>
```

[To better distinguish inputs from outputs, this version of `run` prints a question mark when expecting an input.]

Task. Write a `GP`-program `add` that reads two integers, writes the sum of the two integers, and ultimately returns `()`.

Task. Write a `GP`-program `accum` that reads an integer. If the integer is `0`, it returns the current total. If the integer is not `0`, it adds the integer to the current total, prints the current total, and starts from the beginning.

Task. Instead of running a `GP`-program in the `IO` monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

```
simulate :: GP a -> [Int] -> (a, [Int])
```

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated.

A map function for `GP` can be defined as follows:

```
instance Functor GP where
  fmap f (End x)   = End (f x)
  fmap f (Get g)   = Get (fmap f . g)
  fmap f (Put n x) = Put n (fmap f x)
```

Task. Define sensible instances of `Monad` and `MonadState` for `GP`. How is the behaviour of the `MonadState` instance for `GP` different from the usual `State` type?

Exercise 2.9 (Idiom, continuation passing). Find Haskell definitions for the functions `start`, `stop`, `store`, `add` and `mul` such that you can embed a stack-based language into Haskell:

```
p1, p2, p3 :: Int
p1 = start store 3 store 5 add stop
p2 = start store 3 store 6 store 2 mul add stop
p3 = start store 2 add stop
```

Here, p_1 should evaluate to `8` and p_2 should evaluate to `15`. The program p_3 is allowed to fail at runtime.

Once you have that, try to find a solution that rejects programs that require non-existing stack elements during type checking.

Hint: Type classes are *not* required to solve this assignment. This is somewhat related to continuations. Try to first think about the types that the operations should have, then about the implementation.

3 Monads

The goal of this task is to create your own *monad*. In fact, you have to extend the well-known state monad (see `Control.Monad.State` in the hierarchical libraries), and add some extra features to it.

The first step is to introduce a type constructor for the new monad:

```
data StateMonadPlus s a = ...
```

The type variables `s` and `a` have the standard meaning: `s` is the type of the state to carry and `a` is the type of the return value. Make the new monad an instance of the `MonadState` type class. Hence, you have to include:

```
import Control.Monad.State
```

We now discuss the three additional features that your monad has to support.

Feature 1: Diagnostics

We want to gather information about the number of calls to all primitive functions that work on a `StateMonadPlus`. For this, you have to write a function `diagnostics` with the following type:

```
diagnostics :: StateMonadPlus s String
```

This function should count the number of binds (≫=) and `returns` (and other *primitive* functions) that have been encountered, including the call to `diagnostics` at hand. Secondly, provide a function

```
annotate :: String -> StateMonadPlus s a -> StateMonadPlus s a
```

which allows a user to annotate a computation with a given label. The functions for Features 2 and 3, as well as `get` and `put`, should also be part of the diagnosis.

As an example, consider the input

```
do return 3 ≫ return 4
   return 5
   diagnostics
```

which should return the string

```
"[bind=3, diagnostics=1, return=3]"
```

Note that ≫ is implemented in terms of ≫=, and thus also counts as a bind.
Here is another example:

```
do annotate "A" (return 3 ≫ return 4)
   return 5
   diagnostics
```

This returns the string

```
"[A=1, bind=3, diagnostics=1, return=3]"
```

Feature 2: Failure

A second feature of your monad is that it can fail during a computation. The `Monad` type class offers the following member function:

```
fail :: (Monad m) ⇒ String -> m a
```

Calling this function should not result in an exception. To facilitate this, the function `runStateMonadPlus` (which will be explained later) returns an `Either` value: `Left` indicates that the computation failed, `Right` indicates success. You may have to change the `StateMonadPlus` data type to cope with this.

Feature 3: History of states

The last feature is to save the current state, and to restore a previous state as the current state. Include the following type class definition in your code, and make `StateMonadPlus` an instance of this type class.

```
class MonadState s m ⇒ StoreState s m | m -> s where
  saveState :: m ()
  loadState :: m ()
```

The part `m -> s` in the class declaration is a *functional dependency*, indicating that the type `s` is uniquely determined by the choice of `m`. Functional dependencies limit the instance declarations that are valid, and in turn allow the type checker to make use of the functional dependency while inferring type. Functional dependencies are a Haskell language extension. To enable it, you must put a language pragma at the top of your module:

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

Saving and loading states should be implemented as a stack: saving a state means pushing the current state on the stack, while loading a state means popping a state from the stack to replace the current one. If `loadState` is called with an empty stack, then the computation in the monad should fail (as explained for Feature 2).

Here is an example expression:

```
do i1 <- get; saveState
   modify (*2)
   i2 <- get; saveState
   modify (*2)
   i3 <- get; loadState
   i4 <- get; loadState
   i5 <- get
   return (i1, i2, i3, i4, i5)
```

This program should return the value `(1, 2, 4, 2, 1)` if we start with the state consisting of the integer `1`.

Running the monad

You have to write a function

```
runStateMonadPlus :: StateMonadPlus s a -> s -> Either String (a, s)
```

for running the monad. Given a computation in the `StateMonadPlus` and an initial state, `runStateMonadPlus` returns either an error message if the computation failed, or the result of the computation and the final state.

To turn your module into a proper library, you should also think about which functions should be exposed to outside this module, and which functions should be hidden (and only be visible inside the current module). You might want to consider re-exporting all functionality offered by the `Control.Monad.State` module.

Try also to define a number of unit tests or even QuickCheck properties.

Bonus questions

Exercise 3.1. Do the monad laws hold for `StateMonadPlus`? Explain your answer.

Exercise 3.2. What are the advantages of hiding (constructor) functions? How important is this for each of the three additional features supported by `StateMonadPlus`?

Exercise 3.3. What are the modifications required to make a monad transformer for `StateMonadPlus`?

Exercise 3.4. Suppose that we want to write a function

```
diagnosticsFuture :: StateMonadPlus s String
```

which provides information about the computations in `StateMonadPlus` that are still to come. Explain how this would affect your code. If you feel that such a facility cannot be implemented, then you should give some arguments for your opinion. If you believe it can be done, then try to do so.

4 Programming jointly with types and values

Exercise 4.1 (Fixpoint)**.** The lambda term

```
y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

(that encodes a fixed point combinator in the untyped lambda calculus) does not type check in Haskell. Try it! Interestingly though, recursion on the type level can be used to introduce recursion on the value level. If we define the recursive type

```
data F a = F {unF :: F a -> a}
```

then we can "annotate" the definition of `y` with applications of `F` and `unF` such that `y` typechecks. Do it!

Exercise 4.2 (Nested datatype)**.** Here is a nested datatype for square matrices:

```
type Square a    = Square' Nil a
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a    = Nil
data Cons t a = Cons a (t a)
```

Give Haskell code that represents the following two square matrices as elements of the `Square` datatype:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Note: you don't have to define any functions for the `Square` datatype. Defining sensible functions for `Square` (even `show`) is not entirely trivial and might be the topic of a future assignment.

Exercise 4.3 (Nested datatype)**.** Recall the datatype of square matrices from Assignment 3:

```
type Square      = Square' Nil
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a    =  Nil
data Cons t a =  Cons a (t a)
```

Note that I have eta-reduced the definition of `Square`. This turns out to be necessary in the end where I'll mention it again.

Let's investigate how we can derive an equality function on square matrices. We do so very systematically by deriving an equality function for each of the four types. We follow a simple, yet powerful principle: type abstraction corresponds to term abstraction, and type application corresponds to term application.

What does this mean? If a type f is parameterized over an argument a, then in general, we have to know how equality is defined on a in order to define equality on f a. Therefore we define

```
eqNil :: (a -> a -> Bool) -> (Nil a -> Nil a -> Bool)
eqNil eqA Nil Nil = True
```

In this case, the a is not used in the definition of Nil, so it is not surprising that we do not use eqA in the definition of eqNil. But what about Cons? The datatype Cons has two arguments t and a, so we expect two arguments to be passed to eqCons, something like

```
eqCons eqT eqA (Cons x xs) (Cons y ys) = eqA x y && ...
```

But what should the type of eqT be? The t is of kind * -> *, so it can't be t -> t -> Bool. We can argue that we should use t a -> t a -> Bool, because we use t applied to a in the definition of Cons. However, a better solution is to recognise that, being a type constructor of kind * -> *, an equality function on t should take an equality function on its argument as a parameter. And, moreover, it does not matter what this parameter is! A function like eqNil is polymorphic in type a, so let us require that eqT is polymorphic in the argument type as well:

```
eqCons :: (∀b . (b -> b -> Bool) -> (t b -> t b -> Bool)) ->
          (a -> a -> Bool) ->
          (Cons t a -> Cons t a -> Bool)
eqCons eqT eqA (Cons x xs) (Cons y ys) = eqA x y && eqT eqA xs ys
```

Now you can see how we apply eqT to eqA when we want equality at type t a – the type application corresponds to term application.

Task. A type with a ∀ on the inside requires the extension RankNTypes to be enabled. Try to understand what the difference is between a function of the type of eqCons and a function with the same type but the ∀ omitted. Can you omit the ∀ in the case of eqCons and does the function still work?

Now, on to Square'. The type of eqSquare' follows exactly the same idea as the type of eqCons:

```
eqSquare' :: (∀b . (b -> b -> Bool) -> (t b -> t b -> Bool)) ->
             (a -> a -> Bool) ->
             (Square' t a -> Square' t a -> Bool)
```

We now for the first time have more than one constructor, so we actually have to give multiple cases. Let us first consider comparing two applications of Zero:

```
eqSquare' eqT eqA (Zero xs) (Zero ys) = eqT (eqT eqA) xs ys
```

Note how again the structure of the definition follows the structure of the type. We have a value of type t (t a). We compare it using eqT, passing it an equality function for values of type t a. How? By using eqT eqA.

The remaining cases are as follows:

```
eqSquare' eqT eqA (Succ xs) (Succ ys) = eqSquare' (eqCons eqT) eqA xs ys
eqSquare' eqT eqA _ _                 = False
```

The idea is the same – let the structure of the recursive calls follow the structure of the type.

Task. Again, try removing the ∀ from the type of `eqSquare'`. Does the function still typecheck? Try to explain!

Now we're done:

```
eqSquare :: (a -> a -> Bool) -> Square a -> Square a -> Bool
eqSquare = eqSquare' eqNil
```

Test the function. We can now also give an `Eq` instance for `Square` – this requires the minor language extension `TypeSynonymInstances`, because for some stupid reason, Haskell 98 does not allow type synonyms like `Square` to be used in instance declarations:

```
instance Eq a => Eq (Square a) where
  (==) = eqSquare (==)
```

Task. Systematically follow the scheme just presented in order to define a `Functor` instance for square matrices. I.e., derive a function `mapSquare` such that you can define

```
instance Functor Square where
  fmap = mapSquare
```

This instance requires `Square` to be defined in eta-reduced form in the beginning, because Haskell does not allow partially applied type synonyms.

Exercise 4.4 (Nested datatype)**.** Haskell does not allow partially applied type synonyms.

Recall that in previous assignments, we defined

```
type Square = Square' Nil
```

and not

```
type Square a = Square' Nil a
```

With the former definition (and enabled `TypeSynonymInstances`), we can make `Square` an instance of class `Functor`, with the latter definition, we cannot.

Why is this restriction in place? Try to find problems arising from partially applied type synonyms, and describe them (as concisely as possible) with a few examples.

5 Programming with classes

Exercise 5.1 (Split). Consider the following class

```
class Splittable a where
  split :: a -> (a, a)
```

for types that allow values to be split. Random number generators (for instance `StdGen`) allow such a split operation:

```
instance Splittable StdGen where
  split = System.Random.split
```

We can also make other types an instance of `Splittable`. Define an instance `Splittable [a]` where, assuming that the list passed is infinite, the list is split into one list containing all the odd-indexed elements, and one containing all the even-indexed elements of the original list.

Define an instance `Splittable Int` where `n` is split into `2 * n` and `2 * n + 1`.

Consider the datatype

```
data SplitReader r a = SplitReader {runSplitReader :: r -> a}
```

which is isomorphic to the `Reader` datatype. Define a variant of the `Reader` monad

```
instance (Splittable r) => Monad (SplitReader r)
```

where the passed state is split before it is passed on. Also implement the instance of `MonadReader`:

```
instance (Splittable r) => MonadReader r (SplitReader r)
```

You have to pass enable the `FlexibleInstances` and `MultiParamTypeClasses` language extensions to make GHC accept this instance. The methods of the class `MonadReader` are

```
ask :: (MonadReader r m) => m r
```

that allows you to access the read state, and

```
local :: (MonadReader r m) => (r -> r) -> m a -> m a
```

that allows you to locally modify the read state.

Finally, consider the function

```
labelTree :: Int -> SplitReader Int (Tree Int)
labelTree 0 = return Leaf
labelTree n = return () >> liftM3 Node (labelTree (n - 1)) ask (labelTree (n - 1))
```

where

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
  deriving Show
```

When calling `runSplitReader (labelTree 3) 1`, the function returns

```
Node (Node (Node Leaf 214 Leaf) 54  (Node Leaf 886  Leaf))
     14
     (Node (Node Leaf 982 Leaf) 246 (Node Leaf 3958 Leaf))
```

Is this what you expected? If you remove `return () ≫` in the definition of `labelTree` and try again, what happens? What do these results imply?

Exercise 5.2 (Context reduction)**.** Consider the following system of classes.

```
class A a
class (A a) ⇒ B a
instance A Bool
instance B Bool
instance A a ⇒ A (Maybe a)
instance (A a, B a) ⇒ A [a]
```

Prove `B Int` $\Vdash$ `A (Maybe (Maybe Int))` and $\emptyset \Vdash$ `A (Maybe [Bool])` using the general entailment rules as well as (super) and (inst) from Slides 10-11 to 10-15. Draw proof trees such as on Slide 10-17.

Exercise 5.3 (Evidence translation)**.** Consider the following module:

```
import Control.Monad.Reader
import System.Random

one :: Int
one = 1

two :: Int
two = 2

randomN :: (RandomGen g) ⇒ Int -> g -> Int
randomN n g = (fst (next g) ‘mod‘ (two * n + one)) - n

sizedInt = do
             n <- ask
             g <- lift ask
             return (randomN n g)
```

What is the most general type of `sizedInt`? (Note that type inference may not work for `sizedInt`, but you should be able to explain the error message by now and know how to fix it. Note further that the most general type will only be accepted by GHC in a type signature when `FlexibleContexts` are enabled.)

Assuming this most general type, perfom an evidence translation for all the overloading involved in the functions `randomN` and `sizedInt`. First, define the record types for the classes involved. You can *ignore* the fact that literals and arithmetic operations are overloaded and just use `one` and `two` as monomorphic integers. You only have to include those methods in the records that are actually used in the program above. Hoever, you *should* consider the desugaring (you may simplify and ignore the `let` statements for the patterns) of the `do` notation to the monad operations, as well as the overloaded `ask` and `lift` functions. In order to define the record types correctly, you must enable the `PolymorphicComponents` or `RankNTypes` language extensions to allow polymorphic fields in datatypes.

Then translate `randomN` and `sizedInt` similar to the translation on Slide 10-21. You are allowed to introduce local abbreviations using `let` and `where` for often-used expressions. The resulting program must, of course, still be type correct in Haskell.

6 Type extensions

Exercise 6.1 (GADT)**.** Here is a datatype of contracts:

```
data Contract :: * -> * where
  Pred :: (a -> Bool) -> Contract a
  Fun  :: Contract a -> Contract b -> Contract (a -> b)
```

The datatype is a Generalized Algebraic Datatype (GADT) as will be introduced in the lecture on Monday, 29 March. The constructors do not both target any contract `Contract a`, but the `Fun` constructor has a restricted result type, i.e., it can only construct function contracts. GADTs require the language flag/pragma `GADTs`.

A contract can be a predicate for a value of arbitrary type. For functions, we offer contracts that contain a precondition on the arguments, and a postcondition on the results.

Contracts can be attached to values by means of `assert`. The idea is that `assert` will cause run-time failure if a contract is violated, and otherwise return the original result:

```
assert :: Contract a -> a -> a
assert (Pred p)       x = if p x then x else error "contract violation"
assert (Fun pre post) f = assert post . f . assert pre
```

For function contracts, we first check the precondition on the value, then apply the original function, and finally check the postcondition on the result. Note that the case for `Fun` makes use of the fact that the `Fun` constructor targets only function contracts. Because of this knowledge, GHC allows us to apply `f` as a function.

For example, the following contract states that a number is positive:

```
pos :: (Num a, Ord a) => Contract a
pos = Pred (>0)
```

We have

```
assert pos 2 ≡ 2
assert pos 0 ≡ ⊥       (contract violation error)
```

Task. Define a contract

```
true :: Contract a
```

such that for all values `x`, the equation `assert true x ≡ x` holds. Prove this equation using equational reasoning.

Often, we want the postcondition of a function to be able to refer to the actual argument that has been passed to the function. Therefore, let us change the type of `Fun`:

```
DFun :: Contract a -> (a -> Contract b) -> Contract (a -> b)
```

The postcondition now depends on the function argument.

Task. Adapt the function `assert` to the new type of `DFun`.

Task. Define a combinator

```
(-▹) :: Contract a -> Contract b -> Contract (a -> b)
```

that reexpresses the behaviour of the old `Fun` constructor in terms of the new and more general one.

Task. Define a contract suitable for the list index function `(!!)`, i.e., a contract of type

```
Contract ([a] -> Int -> a)
```

that checks if the integer is a valid index for the given list.

Task. Define a contract

```
preserves :: Eq b => (a -> b) -> Contract (a -> a)
```

where `assert (preserves p) f x` fails if and only if the value of `p x` is different from the value of `p (f x)`. Examples:

```
assert (preserves length) reverse "Hello"         ≡ "olleH"
assert (preserves length) (take 5) "Hello"        ≡ "Hello"
assert (preserves length) (take 5) "Hello world" ≡ ⊥
```

Task. Consider

```
preservesPos  = preserves (>0)
preservesPos' = pos -▹ pos
```

Is there a difference between `assert preservesPos` and `assert preservesPos'`? If yes, give an example where they show different behaviour. If not, try to prove their equality using equational reasoning.

We can add another contract constructor:

```
List :: Contract a -> Contract [a]
```

The corresponding case of `assert` is as follows:

```
assert (List c) xs = map (assert c) xs
```

Task. Consider

```
allPos  = List pos
allPos' = Pred (all (>0))
```

Describe the differences between `assert allPos` and `assert allPos'`, and more generally between using `List` versus using `Pred` to describe a predicate on lists. (Hint: Think carefully and consider different situations before giving your answer. What about using the `allPos` and `allPos'` contracts as parts of other contracts? What about lists of functions? What about infinite lists? What about strict and non-strict functions working on lists?)

7 Performance

Exercise 7.1 (Algorithm design)**.** Given the following type signature

```
smooth_perms :: Int -> [Int] -> [[Int]]
```

for a function which returns all permutations of its second argument for which the distance between each two successive elements is at most the first argument.

```
split [] = []
split (x:xs) = (x, xs) : [(y, x:ys) | (y, ys) <- split xs]
perms [] = [[]]
perms xs = [(v:p) | (v, vs) <- split id xs, p <- perms vs]
smooth n (x:y:ys) = abs (y - x) <= n && smooth n (y:ys)
smooth _ _ =        True
smooth_perms :: Int -> [Int] -> [[Int]]
smooth_perms n xs = filter (smooth n) (perms xs)
```

A straightforward solution is to generate all permutations and then to filter out the smooth ones. This however is expensive. A better approach is to build a tree, for which it holds that each path from the root to a leaf correspond to one of the possible permutations, next to prune this tree such that only smooth paths are represented, and finally to use this tree to generate all the smooth permutations from.

Now define this tree data type, a function which maps a list onto this tree, the function which prunes the tree, and finally the function which generates all permutations.

8 Observing: performance, testing, benchmarking

Exercise 8.1 (Profiling, testing, benchmarking)**.** Elaborate on exercise 7.1 by using various observation tools:

- Give a `quickCheck` specification and check, by defining a function `allSmoothPerms`.
- Use the `criterion` package to make and run benchmarks for the given naive solution and your solution, in order to find out whether your solution really gives higher performance.
- Use heap profiles to analyse and draw conclusions about the differences.

Exercise 8.2 (Quickcheck)**.** QuickCheck's `Arbitrary` class is defined as follows

```
class Arbitrary a where
  arbitrary :: Gen a
```

The type `Gen` is defined as

```
newtype Gen a = MkGen {unGen :: StdGen -> Int -> a}
```

(These definitions are from QuickCheck-2. The definitions in QuickCheck-1 are slightly different, but essentially the same. It does not matter which version you use.) Look at the QuickCheck source code for the definition of the monad instance. Assemble an equivalent monad from the `Reader` and `SplitReader` monads or monad transformers.

Define the function `sizedInt :: Gen Int` just using `ask`, `lift` and `System.Random.randomR` (i.e., not using the internal structure of the `Gen` type), such that `sizedInt` generates a random number between `-n` and `n` where `n` is the read integer.

Exercise 8.3 (Profiling)**.** Generate heap profiles for the following functions:

```
rev  = foldl (flip (:)) []
rev' = foldr (\x r -> r ++ [x]) []
```

by using them as function `f` in a main program as follows

```
main = print $ f [1..1000000]
```

(adapt the size of 1000000 according to the speed of your machine to get good results). Interpret and try to explain the results!

Do the same for

```
conc xs ys = foldr (:) ys xs
conc'      = foldl (\k x -> k . (x:)) id
```

with

```
main = print $ f [1..1000000] [1..1000000]
```

(where f is conc or conc').

Finally, have a look at

```
f₁ = let xs = [1..1000000] in if length xs > 0 then head xs else 0
f₂ = if length [1..1000000] > 0 then head [1..1000000] else 0
```

with

```
main = print f
```

(where f is f_1 or f_2).

Remember that the main point of this assignment is not to send in the heap profiles, but to explain them!

Exercise 8.4 (Forcing evaluation)**.** Write a function

```
forceBoolList :: [Bool] -> r -> r
```

that completely forces a list of booleans *without using* `seq`. Note that pattern matching drives evaluation.

Explain why the function forceBoolList has the type as specified above and not

```
forceBoolList :: [Bool] -> [Bool]
```

and why seq is defined as it is, and

```
force :: a -> a
force a = seq a a
```

is useless.

Exercise 8.5 (Type complexity)**.** A curious fact is that Haskell's type system (even that of Haskell without extensions) has exponential space and time complexity. However, the worst case rarely occurs in practice such that the run-time behaviour of the type checker generally is acceptable. Define a family of Haskell expressions such that the type (i.e., the size of the type expression) grows exponentially in the size of the program. Note that if the type is highly repetitive, the type can internally be represented using sharing. However, different type variables cannot be shared. So, to get a truly large type, you have to try to get as many different type variables as possible. If you find yoursolution on the internet explain how it works!

9 Reasoning (inductive, equational)

Exercise 9.1 (Induction over tree)**.** Consider the following definitions:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
  deriving Show

size :: Tree a -> Int
size (Leaf a)   = 1
size (Node l r) = size l + size r

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

```
flatten :: Tree a -> [a]
flatten (Leaf a)   = [a]
flatten (Node l r) = flatten l ++ flatten r

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Prove the following Theorem using equational reasoning:

```
∀(t :: Tree a) . length (flatten t) ≡ size t
```

Note that using the induction principle on trees, it is sufficient to show the following two cases:

```
∀(x :: a) . length (flatten (Leaf x)) ≡ size (Leaf x)
```

and

```
∀(l :: Tree a) (r :: Tree a) .
  ( length (flatten l) ≡ size l
  && length (flatten r) ≡ size r
  ) -> length (flatten (Node l r)) ≡ size (Node l r)
```

To prove the Theorem, you will need to prove the following Lemma first:

```
∀(xs :: [a]) (ys :: [a]) . length (xs ++ ys) = length xs + length ys
```

You may use facts about `(+)` such that `(+)` is associative or that `0` is the neutral element of addition.

Exercise 9.2 (Isomorphism)**.** Show that the following two types are isomorphic if we ignore the presence of undefined values:

```
type A r a = (r, a -> r -> r)
```

and

```
type B r a = Maybe (a, r) -> r
```

In other words, define functions

```
f :: A r a -> B r a
```

and

```
g :: B r a -> A r a
```

such that ∀(x :: B r a) . f (g x) = x and ∀(x :: A r a) . g (f x) = x. Prove these two statements using equational reasoning.

The first proof is an equality between functions. In order to prove that two functions are equal, prove that they return equal results for equal arguments. I.e., instead of the statement given above, prove instead that

```
∀(x :: B r a) (y :: Maybe (a, r)) . f (g x) y = x y
```

For the second proof, you can use *eta-reduction*, another transformation for lambda-terms. If `f` is a function, then `\x -> f x` is equivalent to `f`.

10 IO, Files, and the rest of the world

Exercise 10.1 (IO unsafety). Consider a function of type

```
runIO :: IO a -> a
```

Why is such a function dangerous? (There are several reasons. Try to give example programs that are dangerous or even demonstrate that something strange and unexpected is going on.)

Exercise 10.2 (IO). Formulate why the following program is troublesome in Haskell (look at the types).

```
import System.IO.Unsafe
import Data.IORef

main =
  let x = unsafePerformIO (newIORef [])
  in  do
        writeIORef x "abc"
        ys <- readIORef x
        putStrLn (show (ys :: [Int]))
```

Find out how the ML language family (SML, OCaml, F#) prevents this problem

Describe their approach in relation to Haskell's. Keep the explanation short and precise (no more than 60 words).

Exercise 10.3 (Chat server). Using STM and the `Network` library, write a simple chat server and client. The server should listen on a particular port (say `9595`) for incoming connections. The client should take the hostname to connect to and a nickname as command line arguments and try to connect to that port. Upon connection, the client should register itself and the nickname with the server. The clients should read messages from the user and display messages from the server. The server should broadcast any message received from any client to all clients (mentioning the nickname). The server should also notify clients about new nicknames joining the chat, and about clients who have left the chat.

Do *not* include any other features in the programs. Instead, try to keep the code as short and concise and elegant as possible.

11 Generic Programming

Exercise 11.1 (SYB show & read)**.** Via classes `Show` and `Read` Haskell values can be encoded as strings and read (parsed) back using `show` and `read` respectively. As such `show` and `read` can be used as a poor man's serialization mechanism similar to `encode` and `decode` (from the SYB slides). Look up the implementation of `Show` and `Read` (part of the base package of the Haskell Platform) and re-implement `Show` and `Read` using the `syb` package. Use Quickcheck to test your generic SYB based `show` and `read` implementation against the default `show` and `read`. It may well be that not all corner cases of showing and reading can be covered. If so, explain where and why this happens.

Exercise 11.2 (Generic Deriving show & read)**.** Redo exercise 11.1 for `read` only, using the generic deriving mechanism available in GHC. Be inspired by the module `Generic.Deriving.Show` for the already implemented `show`.