Monad-Based Programming SS 2023

# Assignment 5

Deadline for solutions: 06.07.2023

## Exercise 1 Going Abstract (7 Points)

Recall the proof assistant from Assignment 2, Exercise 1. Your main task here is to adapt your previous code in such a way that the main program has the following type:

```
proofStep :: (MonadError e m, MonadIO m, MonadState ProofState m) => m ProofState
```

for a suitably chosen type of exceptions `e` (so, we are now integrating IO to `proofStep`). That is, you must update your code in such a way that it **only** uses the effects provided by the indicated type classes. You will then need to come up with a concrete instance of `m`, in order to run your code. The facilities of the indicated type classes must be used in the following way:

- `MonadError` is used to throw errors related to erroneous user input or inapplicability of rules.
- `MonadIO` is used to interact with the user.
- `MonadState` is used to store read and update the current proof state (list of goals).

The parameter `m` will have to be instantiated with a monad, obtained by combining monad transformers for state, exceptions and I/O. Note that there is a certain freedom in defining such an instance, related to the fact that applying state transformers in different order need not produce the same result (!) For example, transforming the state monad with the exception transformer is not the same as transforming the exception monad with the state transformer. The first yields

$$(- \times S)^S \mapsto ((- \times S) + E)^S,$$

while the second yields

$$(- + E) \mapsto ((- + E) \times S)^S,$$

Incidentally, this is one of the reasons to use the above abstract type classes instead of concrete monads and monad transformers. More generally, we are thus following the well-known programming principle of *separating interface from implementation.*

For a worked example in this style, consider the snippet in Fig. 1. In mathematical terms, the involed monad is computed as follows: let $T$ be the IO-monad, its exception-transform is $T(- + E)$; the state monad transformer sends $R$ to $(R(- \times S))^S$, and therefore it sends $T(- + E)$ to $(T(- \times S + E))^S$, which is the resulting monad.

You need to implement the following features in your proof assistant (if you did not fully implement the proof assistant previously, this is OK – you only need the interactive loop to function):

(a) a command of the user to print the current list of goals;

(b) a command of the user to quit;

(c) finishing the interaction loop normally, after all goal are proven.

```
{-# LANGUAGE LambdaCase #-}

import Control.Monad.Except (MonadIO(..), MonadError(..), runExceptT)
import Control.Monad.State.Lazy (StateT(runStateT), MonadState(put, get), MonadIO(..))
import Control.Monad ( void )

data Errors = Quit | Show

-- A little demo of a program with multiple effects: exceptions, state and IO
-- The programs keeps requesting the user to enter a string; these are then
-- collected in the internal store. Exception throwing and catching is used for
-- escaping from the loop
errIOStateDemo :: (MonadError Errors m, MonadIO m, MonadState [String] m) => m ()
errIOStateDemo = do
    -- This is a computation w.r.t. the monad "m", which is a polymorphic parameter

    -- liftIO coerces the IO monad to m, using the type constraint "MonadIO m"
    liftIO $ putStr "Enter a line: "

    line <- liftIO getLine
    case line of

        -- throw an exception if the user entered "q" = user is willing to quit
        "q" -> throwError Quit

        -- throw an exception if the user entered "s" = user is willing to see the state
        "s" -> do lines <- get; liftIO $ print lines; throwError Show

        -- proceed with the control flow normally, otherwise
        _   -> return ()

    -- Grab the current state (thanks to "MonadState [String] m")
    state <- get

    -- Update the state (thanks to "MonadState [String] m")
    put $ line : state

    -- recursive call
    errIOStateDemo

    -- exception handling clause
    `catchError` \case

        -- return back to the main loop
        Show -> errIOStateDemo

        -- reraise the exception
        e    -> throwError e

-- In order to use errIOStateDemo, we need to build a concrete monad, so
-- that the interpreter knows how to instantiate 'm'. From the following
-- code the interpreter will deduce that m = StateT [String] (ExceptT Errors IO)
-- i.e. m is the state transformer of the exception transformer of IO
main :: IO ()
main = void (runExceptT (runStateT errIOStateDemo []))
```

Figure 1: Example of multi-effect program.

## Exercise 2 Parsing FOL Formulas (7 Points)

Implement a parser for FOL formulas, specified with the following BNF:

$\langle \textit{fol-term} \rangle$ ::= $\langle \textit{ident} \rangle$
 | $\langle \textit{ident} \rangle$ '(' $\langle \textit{fol-terms} \rangle$ ')'

$\langle$*fol-terms*$\rangle$ ::= $\langle$*empty*$\rangle$
  | $\langle$*term*$\rangle$
  | $\langle$*term*$\rangle$ '`,`' $\langle$*fol-terms*$\rangle$

$\langle$*fol-form*$\rangle$ ::= T
  | F
  | $\langle$*ident*$\rangle$ '`(`' $\langle$*fol-terms*$\rangle$ '`)`'
  | $\langle$*fol-form*$\rangle$ '`/\`' $\langle$*fol-form*$\rangle$
  | $\langle$*fol-form*$\rangle$ '`\/`' $\langle$*fol-form*$\rangle$
  | $\langle$*fol-form*$\rangle$ '`->`' $\langle$*fol-form*$\rangle$

and integrate it to the proof assistant (so that the user, when asked, could enter terms and formulas in this grammar, during the proof process). Since the orignal goals are given statically, the set of active variable names in runtime is known, which can be used to distinguish variable names from functional and predicate symbols.

You can solve this problem in one of two way, at pleasure:

- You can adapt the home-grown parser library, discussed at the lecture: `https://www8.cs.fau.de/ext/teaching/sose2023/mbprog/parsing.hs`, or

- (recommended) to use the native Haskell Parsec parser combinator library, which is an industrial strength tool for parsing with Haskell. For a gentle introduction, the following free chapter of the "Real World Haskell" book is worth exploring: `https://book.realworldhaskell.org/read/using-parsec.html`.

Note that GenParser is again a monad, but a different (more technically involved) one than discussed at the lecture. However, you can still stansfer your intuition about parser combination, using similarity of interfaces: again we have the operation <|> for alternation, many for iterating the parser multiple times, etc.

## Exercise 3 Lifting vs. Maybe (6 Points)

The goal of this excercise is to formalize the foundamental distinction between *divergence* and *abnormal termination.*

The latter is modelled by the *maybe-monad* $TX = X + 1$, which can be defined in any category with binary coproducts and a terminal object.

The former can at least be interpreted in categories where objects are (particular) partially ordered sets (whose underlying partial order we intuitively understand as the information order), and morphisms are (particular) monotone functions. Then we define the corresponding *lifting monad* as follows: $TX = X_\perp = X \uplus \{\perp\}$ and the order on $X_\perp$ as follows: $x \leqslant y$ if $x, y \in X$ and $x \leqslant y$, or $x = \perp$.

Both monads therefore model non-termination. We proceed to idintify a principled distinction between these monads in that we prove that $X_\perp$ is not generally a coproduct of $X$ and 1. To that end:

(a) Prove that for suitable $X$, $X_\perp$ is not a coproduct of $X$ and 1 in the category Cpo of complete partial orders and continuous functions. **Hint:** proof by contradiciton.

(b) Identify binary coproducts in the category $\mathsf{Cpo}^!_\perp$ of pointed complete partial orders and strict continuous functions, i.e. those continuous functions that preserve $\perp$: $f(\perp) = \perp$. Prove the universal properties of coproducts.

(c) Prove that also in $\mathsf{Cpo}^!_\perp$, $X_\perp$ is generally not a coproduct of $X$ and 1.