Functional Programming
Practicals 3: Monads

Shin-Cheng Mu

FLOLAC 2026

1. **The expression evaluator saga**. Open the file `Eval.hs`.

   (a) **Evaluation in environments**. Try to finish the implementation of *eval*. Note that the type Expr contains let-expression (Let) and variables (Var). What effects do you need to evaluate them? Semantics of such expressions involves an environment, which can be defined by

   $$\text{type Env} = \text{List}\ (\text{Name}, \text{Int})\ .$$

   In the file we also have two methods defined:

   $$\begin{aligned} &\mathit{empty}\ :: \text{Env}\ , \\ &\mathit{extend} :: (\text{Name}, \text{Int}) \to \text{Env} \to \text{Env}\ , \end{aligned}$$

   where *empty* denotes an empty environment and *extend* adds a new pair of $(\mathit{variable}, \mathit{value})$ into the environment.

   To locate a value a in a list of type List (a, b), we may use this function from the Prelude:

   $$\mathit{lookup} :: \text{Eq}\ a \Rightarrow a \to \text{List}\ (a, b) \to \text{Maybe}\ b$$

   To begin with, we pretend that all variables in an expression are properly declared by a Let, therefore *lookup* always returns Just. Define *eval*. What effect(s) does *eval* need? Check the classes of monads defined in `Monads.hs` to find out what class of monad provides the method you need.

   (b) **Executing programs**. Run *eval* on some sample expressions. When executing a monadic program you need to choose an actual implementation of a monad, by using its "run" method. What actual implementation of monad will you use? Choose one in `Monads.hs`

   (c) **Failure**. What if we consider the fact that an expression may have unbound variables, and *lookup* may return Nothing? In this case we might want the evaluation to fail. How do we do that, and what effects does *eval* need now?

   (d) **Functions as environments**. Our "environment" for this application is essentially a mapping, that is, a function, from variable names to values. What if, instead of type Env = List (Name, Int), we define

type Env = Name → Maybe Int .

Implement the following two methods *empty* and *extend*, adapting to the new type.

(e) **Multiple exceptions**. Extend the language with an operator for division, that is,

data Expr = ... | Div Expr Expr .

Evaluating an expression, when the denominator evaluates to 0, the evaluation should fail. But now that there are two ways an evaluation may fail, we would want to distinguish between them. Instead of Maybe, we may define a type that offers more information than Nothing in case of failure:

data Except *err a* = Return *a* | Throw *err* ,
data Err = DivByZero | VarNotFound Name ,

What effects will you use now? Extend *eval* to cover the case for Div, see what effects it needs now, and choose a monad implementation to run your program.

(f) **Catching errors**. We assume that we are working on very early computers where integers are 2-bytes long. Therefore, signed numbers are between 32767 and −32768. Extend Err with

data Err = DivByZero | VarNotFound Name | Overflow | Underflow .

If a result of evaluation exceeds 32767, we get an Overflow error; if the result goes below −32768 we get an Underflow error.

Alter *eval* such that when Overflow happens when evaluating any sub-expression, the exception is caught and the value of the sub-expression is 32767; when Underflow happens, the exception is caught, and the value is −32768, and DivByZero is not dealt with. (Well, it doesn't always make sense, but it's just an example.) If the definition is correct, *eval tstExpr01* should be Return 1 and *eval tstExpr02* should be Throw DivByZero (You have to uncomment them). **H**int: I find it easier to make *eval* call an auxiliary function *eval′*, which should be mutually recursive with *eval*.

(g) **Counting additions**. Suppose we are interested in the number of additions (and only additions, not negation and division) performed when evaluating an expression. Can you implement an *eval* that counts the number of additions performed as well as returning the result? What effects do you need?

2. Regarding "fast product" discussed in the lecture. We aim to prove that

$$fastprod\ xs = return\ (prod\ xs) \quad . \tag{1}$$

(a) Consider the following *work*, which is equivalent to the one given in the lecture apart from using if:

work :: [Int] → Maybe Int
work [ ] = *return* 1

$$work\ (x:xs) = \text{if } x == 0 \text{ then } fail \\ \qquad\qquad \text{else } work\ xs \gg\!= \lambda y \rightarrow return\ (x \times y) \ .$$

Prove that

$$work\ xs = \text{if } elem\ 0\ xs \text{ then } fail \text{ else } return\ (prod\ xs) \ , \tag{2}$$

where *prod* is defined in the handouts and *elem* is defined by

$$elem\ y\ [\,] \quad = \text{False} \\ elem\ y\ (x:xs) = x == y \vee elem\ y\ xs \ .$$

Solution: Induction on xs. The case for $xs := [\,]$ is immediate. Consider $xs := x:xs$. We reason:

$$\begin{aligned} & work\ (x:xs) \\ = \ & \text{if } x == 0 \text{ then } fail \\ & \quad \text{else } work\ xs \gg\!= \lambda y \rightarrow return\ (x \times y) \\ = \ & \quad \{ \text{ induction } \} \\ & \text{if } x == 0 \text{ then } fail \\ & \quad \text{else } (\text{if } elem\ 0\ xs \text{ then } fail \text{ else } return\ (prod\ xs)) \gg\!= \lambda y \rightarrow \\ & \qquad return\ (x \times y) \\ = \ & \quad \{ \text{ since } f\ (\text{if } p \text{ then } x \text{ else } y) = \text{if } p \text{ then } f\ x \text{ else } f\ y \ \} \\ & \text{if } x == 0 \text{ then } fail \\ & \quad \text{else if } elem\ 0\ xs \text{ then } fail \gg\!= \lambda y \rightarrow return\ (x \times y) \\ & \qquad\qquad \text{else } return\ (prod\ xs) \gg\!= \lambda y \rightarrow \\ & \qquad\qquad\quad return\ (x \times y) \\ = \ & \quad \{ \text{ monad laws and laws regarding } fail \ \} \\ & \text{if } x == 0 \text{ then } fail \\ & \quad \text{else if } elem\ 0\ xs \text{ then } fail \\ & \qquad\qquad \text{else } return\ (x \times prod\ xs) \\ = \ & \quad \{ \ \} \\ & \text{if } x == 0 \vee elem\ 0\ xs \text{ then } fail \\ & \quad \text{else } return\ (x \times prod\ xs) \\ = \ & \quad \{ \text{ definitions of } elem \text{ and } prod \ \} \\ & \text{if } elem\ 0\ (x:xs) \text{ then } fail \text{ else } return\ (prod\ (x:xs)) \ . \end{aligned}$$

(b) Prove (1) using (2) and the properties of *catch*.

Solution: We reason:

$$\begin{aligned} & fastprod\ xs \\ = \ & \quad \{ \text{ definition of } fastprod \ \} \end{aligned}$$

$$
\begin{aligned}
& \mathit{catch}\ (\mathit{work}\ \mathit{xs})\ (\mathit{return}\ 0) \\
= \quad & \{\ (2)\ \} \\
& \mathit{catch}\ (\text{if}\ \mathit{elem}\ 0\ \mathit{xs}\ \text{then}\ \mathit{fail}\ \text{else}\ \mathit{return}\ (\mathit{prod}\ \mathit{xs}))\ (\mathit{return}\ 0) \\
= \quad & \{\ \text{if-lifting}\ \} \\
& \text{if}\ \mathit{elem}\ 0\ \mathit{xs}\ \text{then}\ \mathit{catch}\ \mathit{fail}\ (\mathit{return}\ 0) \\
& \qquad\qquad \text{else}\ \ \mathit{catch}\ (\mathit{return}\ (\mathit{prod}\ \mathit{xs}))\ (\mathit{return}\ 0) \\
= \quad & \{\ \text{properties of}\ \mathit{catch}\ \} \\
& \text{if}\ \mathit{elem}\ 0\ \mathit{xs}\ \text{then}\ \mathit{return}\ 0 \\
& \qquad\qquad \text{else}\ \ \mathit{return}\ (\mathit{prod}\ \mathit{xs}) \\
= \quad & \{\ \mathit{elem}\ 0\ \mathit{xs} \Rightarrow \mathit{prod}\ \mathit{xs} = 0\ \} \\
& \text{if}\ \mathit{elem}\ 0\ \mathit{xs}\ \text{then}\ \mathit{return}\ (\mathit{prod}\ \mathit{xs}) \\
& \qquad\qquad \text{else}\ \ \mathit{return}\ (\mathit{prod}\ \mathit{xs}) \\
= \quad & \{\ \text{if elimination}\ \} \\
& \mathit{return}\ (\mathit{prod}\ \mathit{xs})\ \ .
\end{aligned}
$$

(c) We needed (2) because we cannot yet prove (1) directly. The reason is that we do not have a rule telling us what happens when *catch* meets (≫=). The following, unfortunately, does *not* hold for reasonable interpretations of failure catching:

$$\mathit{catch}\ \mathit{mx}\ h \gg\!= f \ = \ \mathit{catch}\ (\mathit{mx} \gg\!= f)\ (h \gg\!= f)\ \ . \tag{3}$$

Find a counter-example, when the monad is Maybe, that (3) does not hold.

3. Implementing monads. Solutions to this exercise are all given in `Monads.hs`. However, try implementing your own.

(a) Implement your own reader monad. Define in `Eval.hs`

$$\text{data MyReader}\ \mathit{env}\ a = \text{MyReader}\ (\mathit{env} \to a)\ \ .$$

Try implementing its *return* and (≫=), then *ask* and *local*.

(b) Implement your own state monad. Start with

$$\text{data MyState}\ s\ a = \text{MyST}\ (s \to (a, s))\ \ .$$

Try implementing its *return* and (≫=), then *get* and *put*.

(c) Can you implement a monad that supports both reader and fail?

(d) Can you implement a monad that supports both reader and state?