Functional Programming
Practicals 2: Monads

Shin-Cheng Mu

FLOLAC 2024

1. The Maybe either returns a value or fails, and when it fails we do not know the cause. One would like to have more information in case of failure. Defined in the file `Except.hs` is the following type for exceptions:

   $$\mathbf{data}\ \mathrm{Except}\ e\ a = \mathrm{Return}\ a \mid \mathrm{Throw}\ e\ \ .$$

   (a) Complete the definitions of *return* and $(\gg\!=)$ for Except e.

   (b) Complete the definition of *catchE*:

   $$\mathit{catchE} :: \mathrm{Except}\ e\ a \rightarrow (e \rightarrow \mathrm{Except}\ e\ a) \rightarrow \mathrm{Except}\ e\ a$$

   (c) Back to evaluating expressions. We assume that we are working on very early computers where integers are 2-bytes long. Therefore, signed numbers are between 32767 and -32768. Evaluating an expression may fail in the following ways:

   $$\mathbf{data}\ \mathrm{Err} = \mathrm{DivByZero} \mid \mathrm{Overflow} \mid \mathrm{Underflow}\ \ .$$

   If a result of evaluation exceeds 32767, we get an Overflow error; if the result goes below -32768 we get an Underflow error. And division by zero is still an error.

   Complete the definition of *eval* such that when Overflow happens when evaluating any sub-expression, the value of the sub-expression is 32767; when Underflow happens, the value is -32768, and DivByZero is not dealt with. (Well, it doesn't always make sense, but it's just an example.) Use funtions *add* and div', which raise Overflow and Underflow errors when necessary, in place of (+) and *div*. If the definition is correct, *eval tstExpr00* should be Return 1 and *eval tstExpr01* should be Throw DivByZero. **H**int: *eval* calls an auxiliary funcition eval', which should be mutually recursive with *eval*.

2. (a) Complete the definitions in the file `EvalLet00.hs`.

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

   $$\mathbf{type}\ \mathrm{Env} = \mathrm{Name} \rightarrow \mathrm{Maybe}\ \mathrm{Int}\ \ ?$$

Implement the following two methods:

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

where *empty* denotes an empty environment, and *extend* extends an environment with a (variable, value) pair, and use them in *eval*.

3. **Pseudorandom numbers** are needed in various applications. The following equation shows a classical (while not mathematically ideal) approach to generate sequences of pseudorandom numbers between $[0 \ldots m-1]$:

$$X_{n+1} = (a \times X_n + c) \ \mathbf{mod} \ m \ ,$$

where X_n is the current number, X_{n+1} is the next, and a and c are positive constants.

   (a) To implement a pseudorandom number generator, one may use a state monad that keeps the current number as the state. Complete the definitions in `Rand00.hs`.

   (b) In `Rand00.hs` the constants m, a, and c are given as global variables. In `Rand01.hs` we keep the constants in a Reader monad. Complete the definitions.

4. In `EvalLet01.hs` we consider an Expr evaluator that raises two kinds of exceptions: division by zero, and variable not found. Complete the definition. Start from implementing *eval*, finding out what effects it demands from the monad. Then implement a monad that does support these effects.

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

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

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

$$\begin{aligned}
&\mathit{work} :: [\mathsf{Int}] \to \mathsf{Maybe\ Int} \\
&\mathit{work}\ [\,] \qquad\ = \mathit{return}\ 1 \\
&\mathit{work}\ (x : \mathit{xs}) = \mathbf{if}\ x == 0\ \mathbf{then}\ \mathit{fail} \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{work}\ \mathit{xs} \gg\!= \lambda y \to \mathit{return}\ (x \times y) \ .
\end{aligned}$$

   Prove that

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

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

$$\begin{aligned}
&\mathit{elem}\ y\ [\,] \qquad\ = \mathsf{False} \\
&\mathit{elem}\ y\ (x : \mathit{xs}) = x == y \lor \mathit{elem}\ y\ \mathit{xs} \ .
\end{aligned}$$

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

(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 ($\gg\!=$). The following, unfortunately, does *not* hold for reasonable interpretations of failure catching:

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

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

(d) However, recall

$$\begin{array}{l} (\langle\$\rangle) :: \mathrm{Monad}\ m \Rightarrow (a \to b) \to (m\ a \to m\ b) \\ f\ \langle\$\rangle\ mx = mx \gg\!= (return \cdot f)\ , \end{array}$$

We can demand that

$$f\ \langle\$\rangle\ catch\ mx\ h \;=\; catch\ (f\ \langle\$\rangle\ mx)\ (f\ \langle\$\rangle\ h)\ . \tag{4}$$

Prove (4) when the monad is Maybe.

(e) With ($\langle\$\rangle$), the function *work* can be defined by

$$\begin{array}{ll} work\ [\,] & = return\ 1 \\ work\ (x : xs) & = \mathbf{if}\ x == 0\ \mathbf{then}\ fail\ \mathbf{else}\ (x\times)\ \langle\$\rangle\ work\ xs\ . \end{array}$$

Prove (1) without going through (2), but using (4).