A Counter Monad

Action monads are often implemented by state transformers. Here is a counter monad that illustrates the idea.

The state is the counter value. A state transformer maps an old counter value to a new counter value and a return value.

```haskell
data Counter a = C (Int -> (Int,a))

-- reset the counter
new :: Counter ()
new = C $ \_ -> (0,())

-- increment the counter:
inc :: Counter ()
inc = C $ \n -> (n+1,())
```
A Counter Monad

-- returning the current value of the counter:
get :: Counter Int
get = C $ \ n -> (n,n)

-- return is nop, >>= is sequential execution
instance Monad Counter where
  return r = C $ \ n -> (n,r)
  (>>=) (C f) g = C $ \ n0 -> let (n1,r1) = f n0
                               C g' = g r1
                               in g' n1

The “runtime system” for our counter monad may look like this:

run :: Counter a -> a
run (C f) = snd (f 0)
A Counter Monad: Example of Use

An example “program” using a counter:

```haskell
myprog :: Counter Int
myprog = do new
           inc >> inc >> inc
           c1 <- get
           inc
           c2 <- get
           return (c1*c2)
```

Run the program:

```haskell
run myprog
```

The result is 12.
Counter Monad with Exceptions

An exception is just an ordinary type, e.g.,

\[
data \text{ Exn} = \text{Overflow} \mid \text{Other}
\]

It is the monad that treats exceptions in a special way. An exception is stored at the place of the return value:

\[
data \text{ ECounter a} = \text{EC} (\text{Int} \to (\text{Int, Either a Exn}))
\]

We do this due to the following concerns:

- There is no other good value to return when an exception occurs.
- This does not affect normal return values if we implement the monad operations properly.
Counter Monad with Exceptions

Let’s say inc will overflow if the counter exceeds 3:

\[
\text{inc} :: \text{ECounter} ()
\]
\[
\text{inc} = \text{EC} f \text{ where } f n \mid n \leq 3 = (n+1, \text{Left } () )
\]
\[
\mid \text{otherwise } = (n, \text{Right Overflow})
\]

The monad operators:

\[
\text{instance Monad ECounter where}
\]
\[
\text{return } r = \text{EC } \_ n \rightarrow (n, \text{Left } r)
\]
\[
(\text{EC } f) \_ \_ = g =
\]
\[
\text{EC } \_ n0 \rightarrow \text{let } (n1, r1) = f n0
\]
\[
\text{EC } g' = \text{either } g \text{ throw } r1
\]
\[
\text{in } g' n1
\]
Counter Monad with Exceptions

Where throw is defined as a command that throws an exception:

\[
\text{throw} :: \text{Exn} \rightarrow \text{ECounter} a
\]
\[
\text{throw } e = \text{EC } \backslash n \rightarrow (n, \text{Right } e)
\]

To allow the user to catch and handle exceptions:

\[
\text{catch} :: \text{ECounter } a \rightarrow (\text{Exn} \rightarrow \text{ECounter } a) \rightarrow \text{ECounter } a
\]
\[
\text{catch } (\text{EC } f) h =
\]
\[
\text{EC } \backslash n0 \rightarrow \text{let } (n1, r1) = f n0
\]
\[
\text{EC } g' = \text{either return } h r1
\]
\[
in g' n1
\]
Counter Monad with Exceptions

The runtime system may look like this:

\[
\text{run} :: \text{ECounter} \ a \rightarrow \text{Either} \ a \ \text{Exn} \\
\text{run} \ (\text{EC} \ f) = \text{snd} \ (f \ 0)
\]

A program that throws an exception due to overflow:

\[
\text{errprog} = \text{inc} \gg \text{errprog}
\]

A program that handles an exception:

\[
\text{witprog} = \text{errprog} \ 'catch' \ \_ \rightarrow \text{return} \ ()
\]