Monads

Antonio Hernandez

June 15, 2022

Contents

This document elaborates on the lectures on the topic of *Monads* imparted by Irfan Ali between May 27 and July 7, 2022.

1 Motivating example

Working file: `monads1.hs`

Following Philip Walder's paper *Monads for functional programming*, we start with the implementation of the "calculator" data type `Expr` together with an "evaluation" function, `eval`, as follows:

```
data Expr = Con Int
  | Add Expr Expr
  | Mul Expr Expr
  | Div Expr Expr

eval' :: Expr -> Int
eval' (Con v)   = v
eval' (Add e f) = eval' e + eval' f
eval' (Mul e f) = eval' e * eval' f
eval' (Div e f) = eval' e `div` eval' f
```

Note that both `Expr` and `eval'` are defined recursively.

```
Prelude> :l monads1.hs
[1 of 1] Compiling Main             ( monads1.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> -- Example:
*Main> -- 2 * 3 + 4
*Main>
*Main> eval' (Add (Mul (Con 2) (Con 3)) (Con 4))
10
'*Main> -- This throws an error:
*Main>
*Main> eval' (Div (Con 3) (Con 0))
*Exception: divide by zero
```

We now want to eliminate the exception error due to division by zero, so we define `divSafe` and `evalSafe` .

```
divSafe :: Int -> Int -> Maybe Int
divSafe _ 0 = Nothing
divSafe x y = Just (x `div` y)

evalSafe :: Expr -> Maybe Int
```

A naive declaration of `evalSafe` such as

```
evalSafe :: Expr -> Maybe Int
evalSafe (Con v)   = v
evalSafe (Add e f) = evalSafe e + evalSafe f
evalSafe (Mul e f) = evalSafe e * evalSafe f
evalSafe (Div e f) = evalSafe e `divSafe` evalSafe f
```

would not work.

Problem: How to feed operators +, *, etc. with numbers. (Need to extract numbers out of 'Maybe'.)

Now this works, but is too verbose:

```
evalSafe :: Expr -> Maybe Int
evalSafe (Con v)   = Just v
evalSafe (Add e f) = case evalSafe e of
  Nothing -> Nothing
  Just e' -> case evalSafe f of
    Nothing -> Nothing
    Just f' -> Just (e' + f')
evalSafe (Mul e f) = case evalSafe e of
  Nothing -> Nothing
  Just e' -> case evalSafe f of
    Nothing -> Nothing
    Just f' -> Just (e' * f')
evalSafe (Div e f) = case evalSafe e of
  Nothing -> Nothing
  Just e' -> case evalSafe f of
    Nothing -> Nothing
    Just f' -> e' `divSafe` f'
```

```
Prelude> :r
[1 of 1] Compiling Main             ( monads1.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> evalSafe (Div (Con 3) (Con 0))
Nothing
*Main>
```

```
*Main> evalSafe (Add (Mul (Con 2) (Con 3)) (Con 4))
Just 10
```

To reduce complexity, we introduce:

```
return :: a -> Maybe a
return = Just

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
ma >>= k = case ma of
  Nothing -> Nothing
  Just a  -> k a
```

Now, we can redefine 'evalSafe' in a consise manner:

```
eval :: Expr -> Maybe Int
eval (Con v)   = return v
eval (Add e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' + f')
eval (Mul e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' * f')
eval (Div e f) = eval e >>= \e' -> eval f >>= \f' -> divSafe e' f'
```

and it works:

```
*Main> :r
[1 of 1] Compiling Main             ( monads1.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> eval (Add (Mul (Con 2) (Con 3)) (Con 4))
Just 10
*Main>
*Main> eval (Div (Con 3) (Con 0))
Nothing
```

1.1 Do notation

It is maybe clearer if we use the alternative formatting:

```
eval :: Expr -> Maybe Int
eval (Con v)   = return v
eval (Add e f) =
  eval e >>= \e' ->
```

```
  eval f >>= \f' ->
  return (e' + f')
eval (Mul e f) =
  eval e >>= \e' ->
  eval f >>= \f' ->
  return (e' * f')
eval (Div e f) =
  eval e >>= \e' ->
  eval f >>= \f' ->
  divSafe e' f'
```

which suggests the "do notation" :

```
eval :: Expr -> Maybe Int
eval (Con v)   = return v
eval (Add e f) = do
  e' <- eval e
  f' <- eval f
  return (e' + f')
eval (Mul e f) = do
  e' <- eval e
  f' <- eval f
  return (e' * f')
eval (Div e f) = do
  e' <- eval e
  f' <- eval f
  divSafe e' f'
```

For the remainder of this document we will avoid the 'do' notation.

2 The `Monad` class

Working file: `monads2.hs`

The discussion above motivates the definition of the `Monad` class:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

```
instance Monad Maybe where
  return :: a -> Maybe a
  return = Just

  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  ma >>= k = case ma of
    Nothing -> Nothing
    Just a  -> k a

*Main> :l monads2.hs
[1 of 1] Compiling Main             ( monads2.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> eval (Add (Mul (Con 2) (Con 3)) (Con 4))
Just 10
*Main> eval (Div (Con 3) (Con 0))
Nothing
```

2.1 IO version

Now we get an IO version for free:

```
evalIO :: Expr -> IO Int
evalIO (Con v)   = return v
evalIO (Add e f) = evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' + f')
evalIO (Mul e f) = evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' * f')
evalIO (Div e f) = evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' `div` f')
```

For example, we could enrich this as follows:

```
evalIO :: Expr -> IO Int
evalIO (Con v)   = return v
evalIO (Add e f) =
  putStrLn ("adding " ++ show e ++ " and " ++ show f) >>= \_ ->
  evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' + f')
evalIO (Mul e f) = evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' * f')
```

Note that `>>= \_ ->` can be substituted with `>>` , so that we can write:

```
evalIO :: Expr -> IO Int
evalIO (Con v)   = return v
```

```
evalIO (Add e f) =
  putStrLn ("adding " ++ show e ++ " plus " ++ show f) >>
  evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' + f')
evalIO (Mul e f) =
  putStrLn ("multiplying " ++ show e ++ " times " ++ show f) >>
  evalIO e >>= \e' -> evalIO f >>= \f' -> return (e' * f')
evalIO (Div e f) =
  evalIO f >>= \f' ->
  case f' of
    0 -> putStrLn "Oh! can not divide by zero" >> return 0
    otherwise ->
      putStrLn ("dividing " ++ show e ++ " by " ++ show f) >>
      evalIO e >>= \e' -> return (e' `div` f')
```

```
*Main> :r
[1 of 1] Compiling Main             ( monads2.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> evalIO (Add (Mul (Con 2) (Con 3)) (Con 4))
adding Mul (Con 2) (Con 3) plus Con 4
multiplying Con 2 times Con 3
10
*Main>
*Main> evalIO (Div (Con 3) (Con 0))
Oh! can not divide by zero
0
*Main> evalIO $ Div (Con 6) (Con 2)
dividing Con 6 by Con 2
3
```

2.2 Homework: handle the Either monad

Note: need to give an argument so as to make it kind `* -> *`

```
*Main> :k Either String
Either String :: * -> *
```

```
evalEither :: Expr -> Either String Int
evalEither (Con v)   = return v
evalEither (Add e f) = evalEither e >>= \e' -> evalEither f >>= \f' ->
  return (e' + f')
```

```
evalEither (Mul e f) = evalEither e >>= \e' -> evalEither f >>= \f' ->
  return (e' * f')
evalEither (Div e f) = evalEither e >>= \e' -> evalEither f >>= \f' ->
  divEither e' f'
```

```
Prelude> :r
[1 of 1] Compiling Main             ( monads2.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> evalEither $ Div (Con 6) (Con 2)
Right 3
*Main> evalEither (Div (Con 3) (Con 0))
Left "Divde by zero"
```

3 List monad

This gives a list of all possible pairs:

```
Prelude> [2,3,4] >>= \x -> [7,8,9] >>= \y -> return (x,y)
[(2,7),(2,8),(2,9),(3,7),(3,8),(3,9),(4,7),(4,8),(4,9)]
```

Note it coincides with:

```
Prelude> (,) <$> [2,3,4] <*> [7,8,9]
[(2,7),(2,8),(2,9),(3,7),(3,8),(3,9),(4,7),(4,8),(4,9)]
```

3.1 Parallel between `fmap` and $(>>=)$

```
Prelude> :set -XTypeApplications
Prelude>
Prelude> :t fmap @[]
fmap @[] :: (a -> b) -> [a] -> [b]
Prelude>
Prelude> :t flip ((>>=) @[])
flip ((>>=) @[]) :: (a -> [b]) -> [a] -> [b]
```

With the notation `bind = flip` $(>>=)$, observe the parallel:

```
fmap ::  (a -> b) -> [a] -> [b]
bind ::  (a -> [b]) -> [a] -> [b]
```

Observe:

```
Prelude> fmap (\x -> x^2) [2,3,4]
[4,9,16]
Prelude> fmap (\x -> [x^2]) [2,3,4]
[[4],[9],[16]]
Prelude>
Prelude> [2,3,4] >>= (\x -> [x^2])
[4,9,16]
```

Similar idea, but with pairs of results:

```
Prelude> fmap (\x -> [x*2, x^2]) [2,3,4]
[[4,4],[6,9],[8,16]]
Prelude>
Prelude> [2,3,4] >>= (\x -> [x*2, x^2])
[4,4,6,9,8,16]
```

3.2 Evaluating `Expr` datatypes

Working file: `monads3.hs`

```
type Value = Int

data Expr = Lit Value
  | Add Expr Expr
  | Mul Expr Expr

-- Example:
-- 2 * 3 + 4
-- Add (Mul (Lit 2) (Lit 3)) (Lit 4)

eval :: Expr -> [Value]
eval (Lit e)   = return e
eval (Add e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' + f')
eval (Mul e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' * f')
```

```
Prelude> :l monads3.hs
[1 of 1] Compiling Main             ( monads3.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> eval $ Add (Mul (Lit 2) (Lit 3)) (Lit 4)
[10]
```

3.3 Exercise: obtain all prime numbers less than or equal to n

```
factors :: Int -> [Int]
factors x = [2..div x 2] >>= \y -> if mod x y == 0 then [y] else []

primes :: Int -> [Int]
primes n = [2..n] >>= \x -> if null (factors x) then [x] else []

*Main> primes 50
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

An alternative is to use `guard`

```
*Main> import Control.Monad
*Main Control.Monad> :t guard
guard :: GHC.Base.Alternative f => Bool -> f ()

factors :: Int -> [Int]
factors x = [2..div x 2] >>= (\y -> guard (mod x y == 0) >> return y)

primes :: Int -> [Int]
primes n = [2..n] >>= (\x -> guard (null (factors x)) >> return x)
```

3.4 Exercise: eval with square roots

```
type Value = Float

data Expr = Lit Value
  | Add Expr Expr
  | Mul Expr Expr
  | Sqr Expr

eval :: Expr -> [Value]
eval (Lit e)   = return e
eval (Add e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' + f')
eval (Mul e f) = eval e >>= \e' -> eval f >>= \f' -> return (e' * f')
eval (Sqr e)   = eval e >>= \e' -> [-sqrt e', sqrt e']

> :r
[1 of 1] Compiling Main             ( monads3.hs, interpreted )
```

```
Ok, one module loaded.
>
> eval $ Add (Lit 2) (Sqr (Lit 9))
[-1.0,5.0]
```

4 State monad

Working file: `monads4.hs`

Variable mutation is achieved in Haskell through function application.
`f ::  s -> (a,s)`

- `a` is the type of the value
- `s` is the state

A "state transformation" *is* an element in the set of functions from `s` to `s` . (Thus a state stransformation can be identified with a *Lambda*.)

Let us start by defining the type synonim:

```
type State s a = s -> (a,s)
```

(The name "`State`" is unfortunate. It should be called "`StateModifier`", since it is a lambda.)

Example:

```
f :: State Int String
f n = (show n, n)
```

```
Prelude> :l monads4.hs
[1 of 1] Compiling Main             ( monads4.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> f 7
("7",7)
*Main>
```

Example:

This models the *imperative command* `i++` :

```
inc :: State Int Char
inc x = ('i', x + 1)

*Main> inc 7
('i',8)
```

4.1 Making `State s` a monad.

What can be made a monad is `State s` with `s` fixed, since:
`:kind State String` evaluates to `* -> *`

To be a "certified" monad we would need to define:
`instance Monad (State s) where`

But to be a monad ("certified" or not), we just need to define 'return' and 'bind'.

Return is easy:

```
-- return:
ret :: a -> State s a
ret a = \s -> (a,s)
```

To motivate 'bind', let us look at the:

4.1.1 "eval" example

```
type Value = Float
data Expr = Lit Value
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr
  | Div Expr Expr

-- we want to keep track of the maximum value
eval :: Expr -> State Float Value
eval (Lit v) = \s -> (v, if v > s then v else s)
eval (Add e f) = \s ->
  let
    (v, s')  = eval e s
    (w, s'') = eval f s'
```

```
    vw       = v + w
  in
    (vw, if vw > s'' then vw else s'')
eval (Sub e f) = \s ->
  let
    (v, s')  = eval e s
    (w, s'') = eval f s'
    vw       = v - w
  in
    (vw, if vw > s'' then vw else s'')
eval (Mul e f) = \s ->
  let
    (v, s')  = eval e s
    (w, s'') = eval f s'
    vw       = v * w
  in
    (vw, if vw > s'' then vw else s'')
eval (Div e f) = \s ->
  let
    (v, s')  = eval e s
    (w, s'') = eval f s'
    vw       = v / w
  in
    (vw, if vw > s'' then vw else s'')
```

```
*Main> :r
[1 of 1] Compiling Main             ( monads4.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> -- Let us represent a function with a list of evaluations over a
*Main> -- specified domain:
*Main>
*Main> map (eval $ Mul (Lit 5) (Sub (Lit 7) (Lit 3))) [15..25]
[(20.0,20.0),(20.0,20.0),(20.0,20.0),(20.0,20.0),(20.0,20.0),(20.0,20.0),
(20.0,21.0),(20.0,22.0),(20.0,23.0),(20.0,24.0),(20.0,25.0)]
```

This motivates the definition

```
-- bind:
bind :: State s a -> (a -> State s b) -> State s b
bind sa k = \s -> (k . fst $ sa s) (snd $ sa s)
```

4.2 Another example of Sate

Working file: `monads5.hs`

Let us introduce a lookup table. We add the data constructor `Dec Ident Value` .

```
type Value = Int
type Ident = String

type Table = [(Ident, Value)]

data Expr = Lit Value           -- Literal
  | Var Ident           -- Variable
  | Dec Ident Value     -- Declaration
  | Seq [Expr]          -- Sequence
  | Add Expr Expr
  | Mul Expr Expr
```

The commented tags are to be interpreted as follows:

- `Literal` : We only care about the value ("literal value")
- `Variable` : Variable lookup
- `Declaration` : Variable assignment (set)
- `Sequence` : Sequence of lookups

A value of type `Table` represents the "current state" of assignments of *values* to *variables* (strings).

We want to implement
`eval :: Expr -> Table -> (Value, Table)`
which combines two needs:

- `Table -> Value` e.g. to read the value assigned to a variable (get)
- `Table -> Table` to modify the table, e.g. because a new binding has been made (put)

We use `State` for this need:

```
type State s a = s -> (a,s)

eval :: Expr -> State Table Value
```

We begin to implement `eval` as follows.

```
eval :: Expr -> State State Value
eval (Lit v)   = \t -> (v, t)
eval (Add e f) = \t ->
  let
    (e', t')  = eval e t     -- [1]
    (f', t'') = eval f t'    -- [2]
  in (e' + f', t'')          -- [3]
```

Notes:

- [1] `t'` might defer from `t` e.g. because `eval e t` might be incorporating a declaration.
- [2] We pass `t'` because we want to pass the latest version of the state. Note that `t''` might defer from `t'` (same reason as above).
- [3] We return the latest version of the state (the table).

Multiplication is handled exactly the same:

```
eval (Mul e f) = \t ->
  let
    (e', t')  = eval e t
    (f', t'') = eval f t'
  in (e' * f', t'')
```

Now, how do we implement *declaration*?

```
eval (Dec x v) = \t -> (v, (x, v) : t)
```

Note that we do not care if an assignment `(x,v)` was allready present, because we will define a `lookup` that will only consider the first declaration in the list (and we are disregarding memory efficiency).
And how do we implement `eval` of variable lookup?
One possibility is

```
eval (Var x) = \t ->
  let
    v = head [ v | (x, v) <- t]
  in (v, t)
```

but, since the lookup might fail is preferable to use the built-in `lookup` :

```
Prelude> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
Prelude>
Prelude> -- Example:
Prelude> t = [('a', 1), ('b', 2), ('c', 3), ('b', 4)]
Prelude> lookup 'b' t
Just 2
Prelude> lookup 'd' t
Nothing
```

So we implement lookup as follows:

```
eval (Var x) = \t ->
  case lookup x t of
    Nothing -> error "Undefined variable"
    Just v  -> (v,t)
```

How do we implement `eval` of a sequence? We only care of the evaluation of the last element in the sequence:

```
eval (Seq [])     = error "Empty sequence"
eval (Seq [e])    = eval e
eval (Seq (e:es)) = \t ->
  let
    (_, t') = eval e t
  in eval (Seq es) t'
```

Putting everything togehter, the declaration of `eval` is:

```
eval :: Expr -> State Table Value
eval (Lit v)      = \t -> (v, t)
eval (Var x)      = \t ->
  case lookup x t of
    Nothing -> error "Undefined variable"
```

```
    Just v  -> (v,t)
eval (Dec x v)    = \t -> (v, (x, v) : t)
eval (Seq [])     = error "Empty sequence"
eval (Seq [e])    = eval e
eval (Seq (e:es)) = \t ->
  let
    (_, t') = eval e t
  in eval (Seq es) t'
eval (Add e f)    = \t ->
  let
    (e', t')  = eval e t
    (f', t'') = eval f t'
  in (e' + f', t'')
eval (Mul e f)    = \t ->
  let
    (e', t')  = eval e t
    (f', t'') = eval f t'
  in (e' * f', t'')
```

4.2.1 Example:

A sequence of expressions. The first three assign x, y and z. The last one computes $x + z$.

```
sq1 = Seq [(Dec "x" 7), (Dec "y" 17), (Dec "z" 29),
   (Add (Var "x") (Var "z"))
  ]
```

Another sequence:

```
sq2 = Seq [(Dec "x" 7), (Dec "y" 17), (Dec "z" 29),
   (Add (Mul (Var "x") (Var "z")) (Var "y"))
  ]
```

```
Prelude> :r
[1 of 1] Compiling Main             ( monads5.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> eval sq1 []
(36,[("z",29),("y",17),("x",7)])
*Main>
```

```
*Main> eval sq2 []
(220,[("z",29),("y",17),("x",7)])
```

The "process" is shown in reverse order, so that the head of the list is the desired result.

Now, let's see how all this can be simplified using the State monad.

4.3 Implementation using the `State` monad

Working file: `monads6.hs`

Note that, internally, `State` is declared as

```
newtype State s a = State { runState :: s -> (s, a) }
```

`State` provides the following resources:

```
get ::  State s s
put ::  s -> State s ()
modify ::  (s -> s) -> State s ()
```

Let us implement `eval` . (Note that we have renamed 'Dec' to 'Set'.)

```
import Control.Monad.State

type Value = Int
type Ident = String

type Table = [(Ident, Value)]

data Expr = Lit Value
  | Var Ident
  | Set Ident Value
  | Seq [Expr]
  | Add Expr Expr
  | Mul Expr Expr

eval :: Expr -> State Table Value
eval (Lit v)      = return v
eval (Var x)      = get >>= \t ->
  case lookup x t of
```

```
    Nothing -> error "Variable not defined"
    Just v  -> return v
eval (Set x v)    = modify ((x,v):) >> return v
eval (Seq [])     = error "Empty sequence"
eval (Seq [e])    = eval e
eval (Seq (e:es)) = eval e >> eval (Seq es) >>= return
eval (Add e f)    = eval e >>= \v -> eval f >>= \w -> return (v + w)
eval (Mul e f)    = eval e >>= \v -> eval f >>= \w -> return (v * w)
```

Consider the same sample calculations as before,

```
sq1 = Seq [(Set "x" 7), (Set "y" 17), (Set "z" 29),
  (Add (Var "x") (Var "z"))
 ]

sq2 = Seq [(Set "x" 7), (Set "y" 17), (Set "z" 29),
  (Add (Mul (Var "x") (Var "z")) (Var "y"))
 ]
```

Note that we can not execute `eval sq1 []` as before, since

```
*Main> :t eval sq1
eval sq1 :: State Table Value
```

which is not exactly of type `Table -> (Value, Table)` . For this we use the *getter* `runState` .

```
*Main> :t runState (eval sq1)
runState (eval sq1) :: Table -> (Value, Table)
```

so `runState (eval sq1)` gives what above, i.e. with our "by hand" implementation of `State`, was simply given by `eval sq1` .

So the correct execution of `eval sq1` is

```
*Main> runState (eval sq1) []
(36,[("z",29),("y",17),("x",7)])
*Main> runState (eval sq2) []
(220,[("z",29),("y",17),("x",7)])
```

which gives the same results as above.

4.3.1 Alternative definition of `eval (Set x v)`

We can use `put` instead of `modify` in the declaration of `eval` for `Set` . Instead of

```
eval (Set x v) = modify ((x,v):) >> return v
--                      [1]             [2]
```

we could have written

```
eval (Set x v) = get >>= \t -> put ((x,v):t) >> return v
--               [3]          [4]                  [5]
```

Notes:

[1] `modify ((x,v):) :: (State Table) ()`

[2] `return v :: (State Table) Value`

so that [1] >> [2] is of type `(State Table) Value`

[3] `get :: (State Table) Table`

[4] `\t -> put ((x,v):t) :: State -> (State Table) ()`

so that [3] >>= [4] is of type `(State Table) ()`

[5] `return v :: (State Table) Value`

so that [3] >>= [4] >> [5] is of type `(State Table) Value`

We have parenthesized `(State Table)` to emphasize that *it* is tthe monad.

4.4 Implementation of `State` monad

Working file: `monads6.hs`

For simplicity, we do the implementation using the type synonim (instead of newtype).

```
type State2 s a = s -> (a,s)
```

4.4.1 `return`

```
ret :: a -> State2 s a
ret a = \s -> (a,s)
```

4.4.2 bind

```
bind sa k = \s ->
  let
    (a, s') = sa s
  in k a s'
```

4.4.3 get & put

```
get2 :: State2 s s
get2 = \s -> (s,s)

put2 :: s -> State2 s ()
put2 = \s -> \_ -> ((),s)
```

With these definitions, our previous de claration of `eval` becomes:

```
eval2 :: Expr -> State2 Table Value
eval2 (Lit v)      = ret v
eval2 (Var x)      = get2 `bind` \t ->
  case lookup x t of
    Nothing -> error "Variable not defined"
    Just v  -> ret v
eval2 (Set x v)    = get2 `bind` \t -> put2 ((x,v):t) `bind` \_ -> ret v
eval2 (Seq [])     = error "Empty sequence"
eval2 (Seq [e])    = eval2 e
eval2 (Seq (e:es)) = eval2 e `bind` \_ -> eval2 (Seq es) `bind` ret
eval2 (Add e f)    = eval2 e `bind` \v -> eval2 f `bind` \w -> ret (v + w)
eval2 (Mul e f)    = eval2 e `bind` \v -> eval2 f `bind` \w -> ret (v * w)
```

and we verify it works:

```
*Main> :r
[1 of 1] Compiling Main             ( monads6.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> eval2 sq1 []
(36,[("z",29),("y",17),("x",7)])
*Main>
*Main> eval2 sq2 []
(220,[("z",29),("y",17),("x",7)])
```

which reproduces the same results as before.