TRABAJO PRÁCTICO IV

Meli Sebastián. Rodríguez Jeremías.

ANÁLISIS DE LENGUAJES DE PROGRAMACIÓN

27 de marzo de 2017

Ejercicio 1

Ejercicio 1.a

Dado el constructor de tipos State y la siguiente instancia a la clase mónada:

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

instance Monad State where
    return x = State (\s -> (x, s))
    m >>= f = State (\s -> let (v, s') = runState m s in
                           runState (f v) s')
```

Debemos demostrar que es una instancia válida.

- **Monad 1**

```
      return x >>= f
  =<def de return>
      State (\e->(x,e)) >>= f
  =<def de >>=>
      State (\s -> let (x',s') = runState (State (\e->(x,e))) s
                   in  runState (f x') s')
  =<def de runstate>
      State (\s -> let (x',s') = (\e->(x,e)) s
                   in  runState (f x') s')
  =< aplicación de función >
      State (\s -> let (x',s') = (x,s)
                   in  runState (f x') s')
  =< aplicación de let >
      State (\s ->  runState (f x) s )
  =<*>
      State (\s ->  runState (State p) s )
  =<def de runStateate>
      State (\s -> p s)
  =<extensionalidad>
      State p
  = < * >
      f x

  (*) f x = State p
```

- **Monad 2**

```
      (State h) >>= return
  = < def de >>= y de runState >
      State (\s -> let (x,s') = h s
                   in   runState (return x) s' )
  = < def de return >
      State (\s -> let (x,s') = h s
```

```
                in   runState (State (\e -> (x,e))) s' )
= <def de runStateate y aplicación >
    State (\s -> let (x,s') = h s
                in  (x,s') )
= <extensionalidad>
    State  (\s -> h s)
= < extensionalidad>
    State h
```

- **Monad 3**

```
    (State h >>= f) >>= g
=< >>= y def de runState>
    (State (\s -> let (x,s') = h s in   runState (f x) s' ) ) >>= g
= <*>
    (State (\s -> let (x,s') = (a0,s0) in   runState (f x) s' ) ) >>= g
=<aplico let>
    (State (\s -> runState (f a0) s0 ) ) >>= g
= <*>
    (State (\s -> runState (State kf) s0 ) ) >>= g
= < def de runState >
    (State (\s -> kf s0) ) >>= g
= < >>= y def de runState>
    State (\e -> let (z0,z1) = (\s -> kf s0) e in runState (g z0) z1)
= < aplico >
    State (\e -> let (z0,z1) = kf s0 in runState (g z0) z1)
= <*>
    State (\e -> let (z0,z1) = (a1,s1) in runState (g z0) z1)
= < evaluo let>
    State (\e -> runState (g a1) s1)
= < evaluacion de let >
    State (\s ->    let (p0,p1) = (a1,s1) in runState (g p0) p1    )
= < * >
    State (\s ->    let (p0,p1) = kf s0 in runState (g p0) p1    )
= < runState def >
    State (\s -> runState ( State (\e -> let (p0,p1) = kf e in runState (g p0) p1 )  ) s0)
=< >>= >
    State (\s -> runState ((State kf) >>=g) s0)
= <*>
    State (\s -> runState (f a0 >>=g) s0)
=<aplico let>
    State (\s -> let (x,s') = (a0,s0) in runState (f x >>=g) s')
=<*>
    State (\s -> let (x,s') = h s in runState (f x >>=g) s')
= < app>
    State (\s -> let (x,s') = h s in runState ( (\x'-> f x' >>=g) x) s' )
=< >>= y def de runState>
    State h >>= (\x -> f x >>=g)
```

```
*) h s = (a0,s0)
*) f a0 = (State kf)
*) kf s0 = (a1,s1)
```

Ejercicio 1.b

Ver en Eval1.hs

Ejercicio 2

Ejercicio 2.a

```
newtype StateError a = StateError { runStateError :: Env -> Maybe (a, Env) }

instance Monad StateError where
        return x           = StateError (\e -> Just (x,e))
        StateError h >>= f = StateError (\e -> case (h e) of
                                                 Nothing     -> Nothing
                                                 Just (a,e') -> runStateError (f a) e' )
```

Ejercicio 2.b

```
instance MonadError StateError where
        throw = StateError (\e -> Nothing)
```

Ejercicio 2.c

```
instance MonadState StateError where
    lookfor v = StateError (\s -> Just (lookfor' v s, s))
                where lookfor' v ((u, j):ss) | v == u = j
                                             | v /= u = lookfor' v ss
        -- Suponemos que no se utilizan variables no declaradas en LIS
    update v i = StateError (\s -> Just ((), update' v i s))
                 where update' v i [] = [(v, i)]
                       update' v i ((u, _):ss) | v == u = (v, i):ss
                       update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)
```

Ejercicio 2.d

Ver en Eval2.hs

Ejercicio 3

Ejercicio 3.a

```
newtype StateErrorTick a = StateErrorTick { runStateErrorTick :: Env -> (Maybe (a, Env),Int) }
--Agregamos un Int para contar la cantidad de operaciones. Si se produce un error
--contamos las operaciones hasta el error.
```

Ejercicio 3.b

```
class Monad m => MonadTick m where
        tick :: m ()
```

Ejercicio 3.c

```
instance MonadTick StateErrorTick where
        tick = StateErrorTick (\e -> (Just ( () , e) , 1))
```

Ejercicio 3.d

```
instance MonadError StateErrorTick where
        throw = StateErrorTick (\e -> (Nothing,0))
```

Ejercicio 3.e

```
instance MonadState StateErrorTick where
    lookfor v = StateErrorTick (\s -> (Just (lookfor' v s, s),0) )
                where lookfor' v ((u, j):ss) | v == u = j
                                             | v /= u = lookfor' v ss
        -- Suponemos que no se utilizan variables no declaradas en LIS
    update v i = StateErrorTick (\s -> (Just ((), update' v i s),0))
                 where update' v i [] = [(v, i)]
                       update' v i ((u, _):ss) | v == u = (v, i):ss
                       update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)
```

Ejercicio 3.f

Ver en Eval3.hs