**Universidad Nacional de Rosario**

FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA

**Análisis de Lenguajes de Programación**

R-313

INFORME TRABAJO PRÁCTICO III

Alumnos

Cavigioli, Axel `C-7114/5`
Rassi, Octavio `R-4519/7`
Sferco, Martín `S-5656/1`

7 de Diciembre, 2024

# 1. Enunciado del Ejercicio 1.

**Enunciado.** Demostrar que `State` es efectivamente una mónada.

Para demostrar que `State` es una mónada, debemos dar su instancia de la clase `Monad` y probar que se verifican las tres leyes monádicas para la instancia dada. Recordemos que la definición de `State` y su instancia vienen dadas y son las siguientes:

```haskell
newtype State a = State { runState :: Env -> Pair 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')
```

Luego, únicamente resta probar que se cumplen las siguientes leyes,

| | |
|---|---|
| ⟨ **Monad.1** ⟩ | `return x >>= k  = k x` |
| ⟨ **Monad.2** ⟩ | `m >>= return    = m` |
| ⟨ **Monad.3** ⟩ | `(m >>= k) >>= h = m >>= (\x -> k x >>= h)` |

# 2. Demostración.

Como anticipamos, probaremos individualmente la validez de cada una de las leyes de mónadas para luego concluir que `State` es una mónada.

## 2.1. Monad 1

Desarrollando el lado izquierdo de la igualdad,

```
  return x >>= m
= ⟨ return.1 ⟩
  State (\s -> (x :!: s)) >>= m
= ⟨ >>=.1 ⟩
  State (\s -> let (v :!: s') = runState (State (\s -> (x :!: s))) s
               in  runState (m v) s')
= ⟨ Definición del operador . ⟩
  State (\s -> let (v :!: s') = (runState . State) (\s -> (x :!: s)) s
               in  runState (m v) s')
= ⟨ Lema ⟩
  State (\s -> let (v :!: s') = id (\s -> (x :!: s)) s
               in  runState (m v) s')
= ⟨ id.1 ⟩
  State (\s -> let (v :!: s') = (\s -> (x :!: s)) s
               in  runState (m v) s')
= ⟨ Aplicación ⟩
  State (\s -> let (v :!: s') =  (x :!: s)
               in  runState (m v) s')
= ⟨ Definición de let, igualdad de tuplas ⟩
  State (\s -> runState (m x) s)
= ⟨ η-equivalencia ⟩
  State (runState (m x))
= ⟨ Definición del operador . ⟩
  (State . runState) (m x)
= ⟨ Lema ⟩
  id (m x)
= ⟨ id.1 ⟩
  m x
```

que es el lado derecho de la igualdad, como queríamos probar.

## 2.2. Monad 2

Desarrollando el lado izquierdo de la igualdad obtenemos

```
  m >>= return
= ⟨ >>=.1 ⟩
  State (\s -> let (v :!: s') = runState m s
                in  runState (return v) s')
= ⟨ return.1 ⟩
  State (\s -> let (v :!: s') = runState m s
                in  runState (State (\s -> (v :!: s))) s')
= ⟨ Definición del operador . ⟩
  State (\s -> let (v :!: s') = runState m s
                in  (runState . State) (\s -> (v :!: s)) s')
= ⟨ Lema ⟩
  State (\s -> let (v :!: s') = runState m s
                in  id (\s -> (v :!: s)) s')
= ⟨ id.1 ⟩
  State (\s -> let (v :!: s') = runState m s
                in  (\s -> (v :!: s)) s')
= ⟨ Aplicación ⟩
  State (\s -> let (v :!: s') = runState m s
                in  (v :!: s'))
= ⟨ Definición de let ⟩
  State (\s -> runState m s)
= ⟨ η-equivalencia ⟩
  State (runState m)
= ⟨ Definición del operador . ⟩
  (State . runState) m
= ⟨ Lema ⟩
  id m
= ⟨ id.1 ⟩
  m
```

que es el lado derecho de la igualdad, como queríamos probar.

## 2.3. Monad 3

Desarrollando el lado izquierdo de la igualdad obtenemos

```
m >>= (\x -> k x >>= h)
```

= ⟨ `>>=.1` ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  runState ((\x -> k x >>= h) v) s')
```

= ⟨ Aplicación ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  runState (k v >>= h) s')
```

= ⟨ `>>=.1` ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  runState (State
                 (\s -> let (v' :!: s'') = runState (k v) s
                        in  runState (h v') s'')) s')
```

= ⟨ Definición del operador **.** ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  (runState . State)
                 (\s ->  let (v' :!: s'') = runState (k v) s
                         in  runState (h v') s'') s')
```

= ⟨ Lema ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  id (\s -> let (v' :!: s'') = runState (k v) s
                           in  runState (h v') s'') s')
```

= ⟨ `id.1` ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  (\s -> let (v' :!: s'') = runState (k v) s
                        in  runState (h v') s'') s')
```

= ⟨ Aplicación ⟩

```
State (\s -> let (v :!: s') = runState m s
             in  let (v' :!: s'') = runState (k v) s'
                 in  runState (h v') s'')
```

= ⟨ Propiedad del `let` ⟩

```
State (\s -> let (v' :!: s'') = let (v :!: s') = runState m s
                                in  runState (k v) s'
             in  runState (h v') s'')
```

= ⟨ Aplicación ⟩

```
State (\s -> let (v' :!: s'') = (\s -> let (v :!: s') = runState m s
                                       in  runState (k v) s') s
             in  runState (h v') s'')
```

= ⟨ `id.1` ⟩

```
State (\s -> let (v' :!: s'') = id (\s -> let (v :!: s') = runState m s
                                          in  runState (k v) s') s
             in  runState (h v') s'')
```

= ⟨ Lema ⟩

```
State (\s -> let (v' :!: s'') = (runState . State)
                                (\s -> let (v :!: s') = runState m s
                                       in  runState (k v) s') s
             in  runState (h v') s'')
```

= ⟨ Definición del operador `.` ⟩

```
State (\s -> let (v' :!: s'') = runState (State
                                (\s -> let (v :!: s') = runState m s
                                       in  runState (k v) s')) s
             in  runState (h v') s'')
```

$= \langle$ `>>=.1` $\rangle$

```
State (\s -> let (v :!: s') = runState m s
             in  runState (k v) s') >>= h
```

$= \langle$ `>>=.1` $\rangle$

```
(m >>= k) >>= h
```

lo que concluye la demostración de `Monad.3`. Finalmente, habiendo probado la validez de las tres leyes, queda demostrado que `State` es una mónada. A continuación, enunciaremos los lemas que utilizamos a lo largo de las pruebas.

### 2.3.1. Lema para la composición de `State` y `runState`.

**Lema.** Se verifíca que

`State . runState` $=$ `id` $=$ `runState . State`

**Demostración.** La prueba será por extensión.

Sean `x :: State a`, `g :: Env -> Pair a Env` para algún `a`.

Notemos que, como `State` tiene un único constructor, `x` será de la forma `State f` para algún `f :: Env -> Pair a Env`. Veamos entonces que,

```
(State . runState) x
```

$= \langle$ Definición del operador . $\rangle$

```
State (runState x)
```

$= \langle$ `x = State f` $\rangle$

```
State (runState (State f))
```

$= \langle$ `runState.1` $\rangle$

```
State f
```

$= \langle$ `x = State f` $\rangle$

```
x
```

$= \langle$ `id.1` $\rangle$

```
id x
```

```
(runState . State) g
```

$= \langle$ Definición del operador . $\rangle$

```
runState (State g)
```

$= \langle$ `runState.1` $\rangle$

```
g
```

$= \langle$ `id.1` $\rangle$

```
id g
```

□

□

### 2.3.2. Propiedad del `let in`

**Propiedad.** Las siguientes expresiones son equivalentes

```
let x = let y = z
        in  f y
in  g x
```

```
let y = z
in  let x = f y
    in  g x
```

siempre que $y \notin FV(g\ x)$, donde las aplicaciones `(f y)` y `(g x)` denotan expresiones que pueden (o no) hacer uso de las variables `x` e `y`, respectivamente.