Trabajo Práctico 4

Juan Cruz De La Torre y Agustín Diaz

06/12/2020

1. a)

```haskell
newtype State a b = State { runState :: Env -> Pair a Env }

instance Monad State where
  return x = State (x :!:)                                                    (1)
  m >>= f = State (\s -> let (v :!: s') = runState m s in runState (f v) s')    (2)

-- monad.1       return x >>= f = f x

return x >>= f
-- por (1) y (2)
= State (\s -> let (v :!: s') = runState (State (x :!:)) s in runState (f v) s')
-- por State inverso a runState
= State (\s -> let (v :!: s') = (x :!:) s in runState (f v) s')
-- reemplazo let-in
= State (\s -> runState (f x) s)
-- abstracción
= State (runState (f x))
-- por State inverso a runState
= f x

-- monad.2       t >>= return = t
t >>= return
-- por (2)
= State (\s -> let (v :!: s') = runState t s in runState (return v) s')
-- por (1)
= State (\s -> let (v :!: s') = runState t s in runState (State (v :!:)) s')
-- por State inverso a runState
= State (\s -> let (v :!: s') = runState t s in (v :!:) s')
-- aplicación de :!:
= State (\s -> let (v :!: s') = runState t s in (v :!: s'))
-- reemplazo let-in
= State (\s -> runState t s)
-- abstracción
= State (runState t)
-- por State inverso a runState
= t

-- Propiedad (3)
let x = let y = f
        in h y
in g x
= let y = f in
    let x = h y
        in g x

-- monad.3       (t >>= f) >>= g = t >>= (\x -> f x >>= g)
(t >>= f) >>= g
-- por (2)
= State (\s -> let (v :!: s') = runState (t >>= f) s in runState (g v) s')
-- por (2)
```

```
= State (\s -> let (v :!: s') = runState
    (State (\s2 -> let (v2 :!: s2') = runState t s2 in runState (f v2) s2')) s
    in runState (g v) s')
-- por State inverso a runState
= State (\s -> let (v :!: s') =
    (\s2 -> let (v2 :!: s2') = runState t s2 in runState (f v2) s2') s
    in runState (g v) s')
-- aplicación lambda
= State (\s -> let (v :!: s') =
    let (v2 :!: s2') = runState t s in runState (f v2) s2')
    in runState (g v) s')
-- por propiedad (3)
= State (\s -> let (v2 :!: s2') = runState t s
    in let (v :!: s') = runState (f v2) s2' in runState (g v) s')        (4)

-- Por otra parte:
t >>= (\x -> f x >>= g)
-- por (2)
State (\s -> let (v2 :!: s2') = runState t s in runState ((\x -> f x >>= g) v2) s2')
-- aplicación lambda
State (\s -> let (v2 :!: s2') = runState t s in runState (f v2 >>= g) s2')
-- por (2)
State (\s -> let (v2 :!: s2') = runState t s
    in runState (State (\s2 -> let (v :!: s') = runState (f v2) s2
                           in runState (g v) s')) s2')
-- por State inverso a runState
State (\s -> let (v2 :!: s2') = runState t s
    in (\s2 -> let (v :!: s') = runState (f v2) s2 in runState (g v) s') s2')
-- por aplicación lambda
State (\s -> let (v2 :!: s2') = runState t s
    in let (v :!: s') = runState (f v2) s2' in runState (g v) s'))       (5)

-- Como (4) y (5) son expresiones iguales, queda demostrado que
(t >>= f) >>= g = t >>= (\x -> f x >>= g)
```