Análisis de Lenguajes de Programación

Trabajo Práctico 4

Bautista Marelli   Francisco Alcácer

Diciembre 2020

1. Ejercicio 1

```haskell
>> (monad.1)
        return x >>= f = f x
>> (monad.2)
        t >>= return = t
>> (monad.3)
        (t >>= f) >>= g = t >>= (\x -> f x >>= g)

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')

-- Demostracion de monad.1:
return x >>= f
-- = { def return }
State (\s -> (x :!: s)) >>= f
-- = { def >>= }
State (\s_ -> let (v :!: s_') = runState (State (\s -> (x :!: s))) s_
              in runState (f v) s_')
-- = { def runState }
State (\s_ -> let (v :!: s_') = (\s -> (x :!: s)) s_ in runState (f v) s_')
-- = { b-redex }
State (\s_ -> let (v :!: s_') = (x :!: s_) in runState (f v) s_')
-- = { def let }
State (\s_ -> runState (f x) s_)
-- = { e-redex }
State (runState (f x))
-- = { State (runState) = id }
f x

-- Demostracion de monad.2:
t >>= return
-- = { def >>= }
State (\s -> let (v :!: s') = runState t s in runState (return v) s')
-- = { def return }
State (\s -> let (v :!: s') = runState t s in runState (State (\s_ -> (v :!: s_))) s')
-- = { def runState }
State (\s -> let (v :!: s') = runState t s in (\s_ -> (v :!: s_)) s')
-- = { b-redex }
State (\s -> let (v :!: s') = runState t s in (v :!: s'))
-- = { def let }
State (\s -> runState t s)
-- = { e-redex }
State (runState t)
-- = { State (runState) = id }
t
```

```haskell
-- Propiedad (*)
let x = let y = f
        in h y
in g x
=
let y = f in let x = h y
             in g x

-- Demostracion de monad.3
-- Trabajamos con (t >>= f) >>= g
(t >>= f) >>= g
-- = { def >>= }
State (\s -> let (v :!: s') = runState (t >>= f) s in runState (g v) s')
-- = { def >>= }
State (\s -> let (v :!: s') = runState (State (\s_ -> let (v' :!: s_') = runState t s_
                                                      in runState (f v') s_')) s
             in runState (g v) s')
-- = { def  runState }
State (\s -> let (v :!: s') = (\s_ -> let (v' :!: s_') = runState t s_
                                      in runState (f v') s_') s
             in runState (g v) s')
-- = { b-redex }
State (\s -> let (v :!: s') = let (v' :!: s_') = runState t s in runState (f v') s_'
             in runState (g v) s')
{-
= { Utilizamos (*)
| Tomamos (v :!: s') = x,
          (v' :!: s_') = y,
          runState t s = f,
          runState (f v') s_' = h y
          runState (g v) s' = g x }
-}
State (\s -> let (v' :!: s_') = runState t s in let (v :!: s') = runState (f v') s_'
                                                 in runState (g v) s')  -- (A)

-- Trabajamos con t >>= (\x -> f x >>= g)
t >>= (\x -> f x >>= g)
-- = { def >>= }
State (\s -> let (v :!: s') = runState t s in runState ((\x -> f x >>= g) v) s')
-- = { b-redex }
State (\s -> let (v :!: s') = runState t s in runState (f v >>= g) s')
-- = { def >>= }
State (\s -> let (v :!: s') = runState t s
             in runState (State (\s_ -> let (v' :!: s_') = runState (f v) s_
                                        in runState (g v') s_')) s')
```

```haskell
-- = { def runState }
State (\s -> let (v :!: s') = runState t s
             in (\s_ -> let (v' :!: s_') = runState (f v) s_
                        in runState (g v') s_') s')
-- = { b-redex }
State (\s -> let (v :!: s') = runState t s in let (v' :!: s_') = runState (f v) s'
                                              in runState (g v') s_') -- (B)

-- Luego vemos que en (A) y (B) llegamos a lo mismo, por lo que podemos decir que:
(t >>= f) >>= g = t >>= (\x -> f x >>= g)
```