Trabajo Práctico N° 4

Análisis de Lenguajes de Programación

Mellino, Natalia Farizano, Juan Ignacio

Ejercicio 1) a.

Monad.1

Queremos probar `return x >>= f = f x`

```
return x >>= f
= <state.1>
State (\ s -> (x :!: s)) >>= f
= <state.2>
State (\ s -> let (v :!: s') = runState (State (\ s -> (x :!: s))) s
= <def runState>
State (\ s -> let (v :!: s') = (s -> (x :!: s)) s
= <beta-redex>
State (\ s -> let (v :!: s') = (x :!: s) in runState (f v) s')
= <def Let>
= State (\ s -> runState (f x) s)
= < eta-redex >
= State (runState (f x))
= < State . runState = Id >
f x
```

Monad.2

Queremos probar: `t >>= return = t`

```
t >>= return
= < t :: State a >
State g >>= return
= <state.2>
State (\ s -> let (v :!: s') = runState (State g) s in runState (return v) s')
= <def runState>
State (\ s -> let (v :!: s') = g s in runState (return v) s')
= <state.1>
= State (\ s -> let (v :!: s') = g s in runState (State (\ s -> (v :!: s))) s')
= <def runState>
= State (\ s -> let (v :!: s') = g s in (\ s -> (v :!: s)) s')
=<beta-redex>
= State (\ s -> let (v :!: s') = g s in (v :!: s'))
= <def let>
= State (\ s -> g s)
```

```
= < eta-redex >
= State g
```

Monad.3

Queremos probar: `(t >>= f) >= g = t >>= (\x -> f x >>= g)`

```
(t >>= f) >>= g
= < t :: State a -> t = State h >
(State h >>= f) >== g
= <state.2>
State (\ s -> let (v :!: s') = runState (State h) s
              in runState (f v) s') >>= g
= <def runState>
State (\ s -> let (v :!: s') = h s
              in runState (f v) s') >>= g
= <state.2>
State (\ z -> let (b :!: z') = runState (State (\ s -> let (v :!: s') = h s
                                                        in (runState (f v)) s')) z
              in runState (g b) z')
= <def runState>
State (\ z -> let (b :!: z') = (\ s -> let (v :!: s') = h s
                                        in (runState (f v)) s') z
              in runState (g b) z')
= <beta-redex>
State (\ z -> let (b :!: z') = (let (v :!: s') = h z
                                 in (runState (f v)) s')
              in runState (g b) z')
= < prop let | x = (b :!: z'), y = (v :!: s'), f = h z, h y = (runState (f v)) s', g x =
State (\ z -> let (v :!: s') = h z in (let (b :!: z') = (runState (f v)) s'
                                        in runState (g b) z'))
= < beta-expand >
= State (\ z -> let (v :!: s') = h z in (\ s -> let (b :!: z') = (runState (f v)) s
                                                  in runState (g b) z') s')
= < def runState >
State (\ z -> let (v :!: s') = h z
              in runState (State (\ s -> let (b :!: z') = (runState (f v)) s
                                          in runState (g b) z')) s')
= < beta-expand >
State (\ z -> let (v :!: s') = h z
              in runState ((\ x -> (State (\ s -> let (b :!: z') = (runState (f x)) s
                                                   in runState (g b) z'))) v) s')
= < state.2 >
State (\ z -> let (v :!: s') = h z
              in runState ((\ x -> f x >>= g) v) s')
= < def runState >
State (\ z -> let (v :!: s') = runState (State h) z
              in runState ((\ x -> f x >>= g) v) s')
= < state.2 >
(State h) >>= (\ x -> f x >>= g)
= < t :: State a -> t = State h >
t >>= (\ x -> f x >>= g)
```