Andrés Collares - 4.867.464-5
Agustín Bologna - 4.819.357-8

Práctico 3

Ejercicio 1

Para investigar si h_1 y h_2 son equivalentes se analizaran sus formas, pasando de *"do notation"* a *binds* >>= .

Primero se tiene que $h1$ es:

```
h1 = f >>= \x -> g3 >>= \z -> return (x,z)
h1 = (g1 >>= \x' -> g2 >>= \y' -> return x') >>= \x ->
     g3 >>= \z -> return (x,z)
```

Dado que guardar el resultado de g_1 en x' y luego pasarlo a x es lo mismo que guardarlo en x la primera vez, entonces:

```
h1 = g1 >>= \x -> g2 >>= \y -> g3 >>= \z -> return (x,z)
```

Ahora reescribiendo h_2 se obtiene

```
h2 = g1 >>= \x -> g2 >>= \y -> g3 >>= \z -> return (x,z)
```

Por lo que h_1 y h_2 son equivalentes.

Como h_1 y h_2 son equivalentes solo falta ver si h_3 también es equivalente. En este caso se determina que no es equivalente a través del siguiente contra ejemplo:

```
g1 :: State Int Int
g1 = do
  state <- put 1
  return get state

g2 :: State Int Int
g2 = do
  state <- modify (+ 1)
  return get state

g3 :: State Int Int
g3 = get
```

Ejercicio 5

a) Definiciones

Para las pruebas se utilizaran las siguientes definiciones:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> t = fmap f t
```

además se cumple que:

```
fmap f x = pure f <*> x
```

`sequenceA` y `traverse` son definidas de la siguiente manera

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (a : as) = (:) <$> a <*> sequenceA as

traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
traverse f [] = pure []
traverse f (x : xs) = (:) <$> f x <*> traverse f xs
```

`tr` es una transformación tal que:

```
tr :: (Applicative f, Applicative g) => f a -> g a

tr (pure x) = pure x
tr (f <*> x) = tr f <*> tr x
```

b) Primera prueba

Primero se probará inductivamente que:

```
tr . sequenceA as = sequenceA . map tr as
```

Paso base:

Tomando `as = []` se tiene que:

```
tr . sequenceA [] = sequenceA . map tr []
tr pure [] = sequenceA [] -- por definición de sequenceA y map
pure [] = pure [] -- por definición de tr y sequenceA
```

Paso inductivo:

Suponiendo que la propiedad se cumple para `as = xs` se probará que se cumple para `as = (x : xs)`.

```
tr . sequenceA (x:xs) = tr ((:) <$> x <*> sequenceA xs)
                      = tr (fmap (:) x <*> sequenceA xs)
                      = tr (pure (:) <*> x <*> sequenceA xs)
                      = tr pure (:) <*> tr (x <*> sequenceA xs)
                      = pure (:) <*> tr x <*> tr . sequenceA xs
                      = pure (:) <*> tr x <*> sequenceA . map tr xs -- HI
                      = fmap (:) (tr x) <*> sequenceA . map tr xs
                      = (:) <$> tr x <*> sequenceA . map tr xs
                      = sequenceA . map tr x:xs
```

Por lo tanto se cumple para toda lista finita `as :: Applicative f => [f a]`.

c) Segunda prueba

Ahora se probará inductivamente que:

```
tr . traverse f as = traverse (tr . f) as
```

Paso base:

Tomando `as = []` se tiene que:

```
tr . traverse f [] = traverse (tr . f) []
tr pure [] = pure [] -- por definición de traverse
pure [] = pure [] -- por definición de tr
```

Paso inductivo:

Suponiendo que la propiedad se cumple para `as = xs` se probará que se cumple para `as = (x:xs)`.

```
tr . traverse f (x:xs) = tr ((:) <$> f x <*> traverse f xs)
                       = tr ((:) <$> f x) <*> tr . traverse f xs
                       = tr ((:) <$> f x) <*> traverse (tr . f) xs -- HI
                       = tr (fmap (:) (f x)) <*> traverse (tr . f) xs
                       = tr (pure (:) <*> f x) <*> traverse (tr . f) xs
                       = tr pure (:) <*> tr . f x <*> traverse (tr . f) xs
                       = pure (:) <*> tr . f x <*> traverse (tr . f) xs
                       = fmap (:) (tr . f x) <*> traverse (tr . f) xs
                       = (:) <$> (tr . f x) <*> traverse (tr . f) xs
                       = traverse (tr . f) (x : xs)
```

Por lo tanto se cumple para toda lista finita `as :: a` tal que `f :: Applicative g => a -> g b`.