Prueba de Oposición

AY1 - Simple

Teodoro Freund

10 de Agosto del 2022

18:00

El Ejercicio 1.20[1] (retocado)

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si `c` es un conjunto y `e` un elemento, la expresión `c e` devuelve `True` sii `e` pertenece a `c`.

[1] En la guía del 1er cuatrimestre del 2022

El Ejercicio 1.20[1] (retocado)

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si `c` es un conjunto y `e` un elemento, la expresión `c e` devuelve `True` sii `e` pertenece a `c`.

(I) Definir la constante `vacio :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.

(III) Definir un conjunto de funciones que contenga infinitos elementos, pero no todos, y dar su tipo. Además, demuestre (con ejemplos en Haskell) que tiene infinitas funciones pero no todas.

[1] En la guía del 1er cuatrimestre del 2022

Contexto

- Materia: Paradigmas de Lenguajes de Programación

Contexto

- Materia: Paradigmas de Lenguajes de Programación
- Programación Funcional, `Haskell`

Contexto

- Materia: Paradigmas de Lenguajes de Programación
- Programación Funcional, `Haskell`
- Primera Parte de la materia:
  - Los alumnos ya retomaron `Haskell`, reforzando su sintáxis y las herramientas que nos provee.
  - Ya practicaron con esquemas de recursión como `fold`.
  - Se encuentran atacando problemas más complejos con Haskell, antes de encarar de lleno los temas más formales de la materia.

¿Por qué?

- Es un ejercicio que ejemplifica muchas cosas de Haskell que no se ven tanto en la materia. Es una buena forma de decir *"Terminamos con Haskell, pero Haskell no terminó"*.
- Si se lo extiende un poco, da para hablar un montón de temas, entre ellos:
  + la semántica de `undefined`,
  + contravarianza,
  + cosas infinitas y Haskell,
  + etc.
- Es un buen ejercicio para dar en clase y analizarlo en profundidad.

El Ejercicio 1.20 (retocado), de nuevo

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si `c` es un conjunto y `e` un elemento, la expresión `c e` devuelve `True` sii `e` pertenece a `c`.

(I) Definir la constante `vacio :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.

(III) Definir un conjunto de funciones que contenga infinitos elementos, pero no todos, y dar su tipo. Además, demuestre (con ejemplos en Haskell) que tiene infinitas funciones pero no todas.

`vacio :: Conj a`

Pizarrón/Editor

```
vacio :: Conj a
```

- Sabemos que el tipo del conjunto tiene que ser de `a`, donde `a` puede ser cualquier tipo

vacio :: Conj a

Pizarrón/Editor

```
vacio :: Conj a
vacio = \e ->
```

- Sabemos que el tipo del conjunto tiene que ser de `a`, donde `a` puede ser cualquier tipo
- Entonces, empecemos a programarlo, y seguro tiene que tomar un elemento y hacer algo con él

`vacio :: Conj a`

Pizarrón/Editor

```haskell
vacio :: Conj a
vacio = \e -> False
```

- Sabemos que el tipo del conjunto tiene que ser de `a`, donde `a` puede ser cualquier tipo
- Entonces, empecemos a programarlo, y seguro tiene que tomar un elemento y hacer algo con él
- Si bien es fácil ver que debería retornar siempre `False`, vale la pena analizar que otra cosa podríamos hacer con un elemento sobre el cual no sabemos nada sobre su tipo, y llegar a la conclusión de que en realidad no podemos hacer casi nada

vacio :: Conj a

Pizarrón/Editor

```
vacio :: Conj a
vacio _ = False
```

- Sabemos que el tipo del conjunto tiene que ser de a, donde a puede ser cualquier tipo
- Entonces, empecemos a programarlo, y seguro tiene que tomar un elemento y hacer algo con él
- Si bien es fácil ver que debería retornar siempre False, vale la pena analizar que otra cosa podríamos hacer con un elemento sobre el cual no sabemos nada sobre su tipo, y llegar a la conclusión de que en realidad no podemos hacer casi nada
- Por último, para ser idiomatico con Haskell, podemos extrar el lambda y cambiarlo por un _ ya que no se usa

`agregar :: Eq a => a -> Conj a -> Conj a`

Pizarrón/Editor

```
agregar :: Eq a => a -> Conj a -> Conj a
```

agregar :: Eq a => a -> Conj a -> Conj a

Pizarrón/Editor

```
agregar :: Eq a => a -> Conj a -> Conj a
agregar a c = \e ->
```

`agregar :: Eq a => a -> Conj a -> Conj a`

Pizarrón/Editor

```
agregar :: Eq a => a -> Conj a -> Conj a
agregar a c = \e -> (e == a) || c e
```

agregar :: Eq a => a -> Conj a -> Conj a

Pizarrón/Editor

```
agregar :: Eq a => a -> Conj a -> Conj a
agregar a c e = (e == a) || c e
```

`agregar :: Eq a => a -> Conj a -> Conj a`

Pizarrón/Editor

```haskell
agregar :: Eq a => a -> Conj a -> Conj a
agregar a c e = (e == a) || c e
```

Pequeña conclusión

Toda esta primer parte nos permite, no solo entender como utilizar las funciones como tipos de datos (objetivo de la práctica), pero además programar ejercicios en Haskell un poco más complejos, y empezar a mostrar detalles no centrales de la materia pero importantes, como buenas prácticas, formas idiomáticas, etc.

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (a -> b)
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Bool -> Bool)
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Bool -> Bool)
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int

Conjunto infinito de funciones

Pizarrón/Editor

```haskell
infinitos :: Conj (Int -> Int)
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f =
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f = True
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla, probemos uno que devuelva constantemente True

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f = f
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla, probemos uno que devuelva constantemente True
- Por lo general, teniendo una función, la única opción es aplicarla

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f = f 0
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla, probemos uno que devuelva constantemente True
- Por lo general, teniendo una función, la única opción es aplicarla, en este caso a un Int, como 0

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f = f 0
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla, probemos uno que devuelva constantemente True
- Por lo general, teniendo una función, la única opción es aplicarla, en este caso a un Int, como 0
- Ahora, tenemos el resultado que es un Int, busquemos alguna propiedad para la cual haya infinitos Trues e infinitos Falses

Conjunto infinito de funciones

Pizarrón/Editor

```
infinitos :: Conj (Int -> Int)
infinitos f = f 0 `mod` 2 == 0
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Podemos probar hacerlo de booleanos en booleanos, pero esto puede traer un problema
- Entonces, de minima tiene que ser un tipo infinito, probemos con alguno conocido, por ejemplo Int
- Empecemos a programarla, probemos uno que devuelva constantemente True
- Por lo general, teniendo una función, la única opción es aplicarla, en este caso a un Int, como 0
- Ahora, tenemos el resultado que es un Int, busquemos alguna propiedad para la cual haya infinitos Trues e infinitos Falses , por ejemplo, podemos discernir según la paridad

Ejemplo de infinitos elementos

Pizarrón/Editor

```
ejemplo :: [Int -> Int]
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones

Ejemplo de infinitos elementos

Pizarrón/Editor

```
ejemplo :: [Int -> Int]
ejemplo = [ \x -> if (x == 0) then 2 else 1 ]
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Y tenemos que crear una lista de funciones que al ser aplicadas a 0, retornen un número par

Ejemplo de infinitos elementos

Pizarrón/Editor

```
ejemplo :: [Int -> Int]
ejemplo = [ \x -> 2, \x -> 4, \x -> 6 ]
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Y tenemos que crear una lista de funciones que al ser aplicadas a 0, retornen un número par , para facilitar el ejercicio, podemos hacer que devuelvan siempre el mismo valor.

Ejemplo de infinitos elementos

Pizarrón/Editor

```
ejemplo :: [Int -> Int]
ejemplo = [ const 2, const 4, const 6 ]
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Y tenemos que crear una lista de funciones que al ser aplicadas a 0, retornen un número par , para facilitar el ejercicio, podemos hacer que devuelvan siempre el mismo valor. Y un poco más idiomaticamente...

Ejemplo de infinitos elementos

Pizarrón/Editor

```haskell
ejemplo :: [Int -> Int]
ejemplo = map const [2,4..]
```

- Sabemos que el tipo del conjunto tiene que ser un conjunto de funciones
- Y tenemos que crear una lista de funciones que al ser aplicadas a 0, retornen un número par , para facilitar el ejercicio, podemos hacer que devuelvan siempre el mismo valor. Y un poco más idiomaticamente...
- Y para hacerla infinita podemos usar alguna de las muchas facilidades que Haskell nos provee para manejar listas infinitas

Ejemplo de infinitos no-elementos

Pizarrón/Editor

```
contraejemplo :: [Int -> Int]
contraejemplo = map const [1,3..]
```

Decibilidad de la pertenencia al conjunto

Pizarrón/Editor

```
indecidible :: Int -> Int
indecidible x =
```

- Busquemos alguna funcion que sea indecidible verificar su pertenencia

Decibilidad de la pertenencia al conjunto

Pizarrón/Editor

```haskell
indecidible :: Int -> Int
indecidible x = indecidible $ x + 1
```

- Busquemos alguna funcion que sea indecidible verificar su pertenencia
- Una opción es alguna función que nunca termine, de tal forma de nunca poder saber el resultado de aplicarla a 0

Decibilidad de la pertenencia al conjunto

Pizarrón/Editor

```
indecidible :: Int -> Int
indecidible x = indecidible $ x + 1

perteneceindecidible = infinitos indecidible
```

- Busquemos alguna funcion que sea indecidible verificar su pertenencia
- Una opción es alguna función que nunca termine, de tal forma de nunca poder saber el resultado de aplicarla a 0
- Apenas infinitos intente aplicarla a 0 para verificar su pertenencia se va a colgar

Fin

¿Preguntas?