Bitácora

Listas infinitas y lazy evaluation

funcional listas infinitas lazy evaluation

19 de Mayo, 2025

Tarea para la clase que viene:

- Corregir la primera entrega del TP integrador.
- Seguir con la segunda entrega del TP integrador
- Pueden comenzar a realizar parciales para practicar. Les recomendamos:
  - Si todavía no lo hicieron, Tierra de Bárbaros con posible resolución
  - Haskell Chef con posible resolución
  - Padrinos mágicos.
  - Pinky y Cerebro
  - Y todos los que tengan resolución así tienen con qué comparar.

¿Qué vimos hoy?

- Listas infinitas
- Lazy evaluation
- Resolvimos malas prácticas de programación (code smells) con Disfuncional

Listas infinitas

Ya vimos que en Haskell podemos modelar una biblioteca con las listas, por ejemplo:

```
biblioteca = [elVisitante, shingekiNoKyojin1, fundacion, sandman5, brisignr, legado]
```

Y también podemos modelar una lista del 1 al 5:

```
unoAlCinco = [1,2,3,4,5]
```

Pero si quisiéramos hacer una lista del 1 al 1000... ¡¿deberíamos escribir mil veces los números?! 😱 Por suerte, nuestro gran amigo Haskell puede ayudarnos gracias a las *listas por rangos*:

```
unoAlMil = [1..1000]
```

También podemos definir una lista de los números pares entre 1 y 100 de esta forma:

```
paresAlCien = [2,4..100]
```

Y no solo sirve para números, sino también para letras:

```
abecedario = ['a'..'z']
```

Y así como podemos definir listas con límites o con rangos, también podemos tener... ¡**listas infinitas**!

```
infinita = [1..]
```

(¿Lo probaste en la consola y te olvidaste qué hacer para que pare? 😵 Apretá **ctrl + c**. 😉)

Lazy evaluation

Sabemos aplicar la función ´head´ a una lista:

```
head ["hola", "como", "estás?"]
> "hola"
```

Pero, ¿qué pasará con una lista infinita? 😮

```
head [1..]
> 1
```

Por si quedan dudas de qué es lo que acaba de pasar, Haskell no esperó a que terminara la lista sino que tomó directamente lo que necesitaba. Eso es porque su forma de trabajo es la **evaluación perezosa** o **lazy evaluation**. Esto no pasa con todos los lenguajes. Otros (que seguramente ya utilizaste) usan la **evaluación ansiosa** o **eager evaluation** en donde, por ejemplo, esperarían a que la lista termine de cargar (infinitamente nunca 😵) para devolver el primer elemento. Sipi, Haskell es lo más. 😍

Ahora, ¿cómo funciona lazy evaluation? Este tipo de evaluación se basa en una *estrategia* que se llama **call-by-name**... ¿eeehhh? 😳 Simplemente es operar primero las funciones que están "por fuera", antes que las funciones de sus argumentos. Es decir, las funciones se aplican antes de que se evalúen los argumentos. 😎 Si volvemos al ejemplo anterior:

```
head [1..]
-- aplicará primero head, antes que evaluar la lista infinita
> 1
```

Pero también hay funciones en las cuales necesitamos evaluar primero los parámetros, antes que la función en sí:

```
(*) (2+3) 5
(2+3) * 5

-- (*) necesita que sus parámetros sean números para poder evaluar, entonces se evalúa

5 * 5
> 25
```

Evaluar primero los parámetros para luego pasarle el valor final a las funciones, lo llamamos **call-by-value**. Y es la estrategia en la que se basa la eager evaluation. Veamos:

```
head [1..]
-- espera a que termine la lista infinita (nunca 😖)
head [1,2..]
-- espera a que termine la lista infinita (nunca 😖)
head [1,2,3..]
-- espera a que termine la lista infinita (nunca 😖)
head [1,2,3,4..]
-- ... y así hasta el infinito de los tiempos  . ¡No termina!
```

Vimos los siguientes casos teniendo en cuenta estas preguntas:

- ¿terminarán de evaluar con lazy evaluation?

- ¿y con eager evaluation?
- ¿qué nos devuelve?

```
take 15 [1,3..]
-- Sí termina con lazy. No terminaría con eager. Devuelve [1,3,5,7,9,11,13,15,17,19,21,

last [1..]
-- No termina con lazy y tampoco con eager.

length [1..]
-- No termina con lazy y tampoco con eager.

sum [3, 6..3*10]
-- Termina con ambas y devuelve 165.

any even [2, 4..]
-- Termina con lazy pero no con eager. Devuelve True.

all even [2, 4..]
-- No termina

all odd [2, 4..]
-- Devuelve False

head . filter (> 3) $ [1..])
-- Termina con lazy pero no con eager. Devuelve 4.

head . filter (< 3) $ [5..]
-- No termina con lazy y tampoco con eager.

map (*3) [1..]
-- No termina pero devuelve [3, 6, 9…]

fst ("Hola", [1..])
-- Devuelve "Hola". No terminaría de evaluarse con eager.

fst("Hola", head [])
-- Devuelve "Hola". Con eager rompería porque no se puede hacer head de la lista vacía.

snd ([1, "Hola"], 2)
-- Rompe porque las listas deben ser homogéneas.
```

Funciones para generar listas

```
repeat "Hola"
> ["Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hol

iterate (*2) 2
> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, ….]

replicate 10 "Hola"
> ["Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola", "Hola"]

cycle [1, 2, 3]
> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1,
```

Tip para el parcial: funciones para modificar un campo de una estructura

En parciales es muy común que tengamos estructuras de datos complejos y se repita la idea de querer “modificar” uno de los campos de esta estructura. Por ejemplo, si tenemos una persona con nombre, apellido y edad y queremos hacer que cumpla años, duplicar su edad, hacer que cumpla 100 años, etc. Estas funciones terminarían con una lógica muy similar entre sí:

```
cumplirAños :: Persona -> Persona
cumplirAños    unaPersona = unaPersona { edad = edad unaPersona + 1 }

duplicarEdad :: Persona -> Persona
duplicarEdad   unaPersona = unaPersona { edad = edad unaPersona * 2 }

cumplir100Años :: Persona -> Persona
cumplir100Años unaPersona = unaPersona { edad = 100 }
```

¡Esta repetición de lógica la podemos evitar de la misma forma que siempre! Extrayendo la lógica común en una función.

```haskell
modificarEdad :: (Int -> Int) -> Persona -> Persona
modificarEdad unaFuncion unaPersona = unaPersona { edad = unaFuncion . edad $ unaPerson
```

Ahora, podemos escribir nuestras funciones anteriores en función de `modificarEdad`:

```haskell
cumplirAños :: Persona -> Persona
cumplirAños    unaPersona = modificarEdad (+ 1)

duplicarEdad :: Persona -> Persona
duplicarEdad   unaPersona = modificarEdad (* 2)

cumplir100Años :: Persona -> Persona
cumplir100Años unaPersona = modificarEdad (const 100)
```

El crear estas funciones auxiliares nos trae un montón de ventajas:

- Evitamos la repetición de lógica.
- Nos facilita usar composición (en el caso que queramos modificar dos campos distintos a la vez, sólo necesitamos componer dos de estas funciones).
- Agrega una pequeña capa de abstracción entre nuestra lógica de dominio y la estructura de nuestros datos. Esto hace que si nuestra estructura cambia, las únicas funciones que se ven afectadas son las auxiliares, y no las de dominio.

Links Útiles

- Listas infinitas
- Lazy evaluation
- Estrategias de evaluación
- Posible solución de Disfuncional

Podes ver nuestro github acá.
Y saber mas sobre nosotros acá

← Corrección de Entrega 1 TP Funcional

Simulacro de parcial funcional →