Bitácora

Funcional, composición y tipos

funcional tipado inmutabilidad precedencia-de-operadores

07 de Abril, 2025

Tarea para la clase que viene:

- Armar grupo si todavía no lo hiciste. Podés buscar integrantes en el canal **#buscando-grupo** de Discord. Una vez que ya lo tengas armado, anunciarlo en el canal **#grupos** indicando quiénes lo integran.
- Instalarse Haskell **SIN PdePreludat** y Visual Studio Code.
- Resolver el ejercicio PdeP - commerce para ir practicando. Pueden escribir un archivo con extensión `.hs` en el VS Code y luego, en la terminal que pueden abrir ahí mismo, probar lo que hicieron con el comando `stack ghci <archivo>`.

¿Qué vimos hoy?

Paradigma funcional

Es el paradigma con el vamos a arrancar. Y trata sobre... ¡adivinaste! Funciones 😆. Y acá es donde hacemos esa gran pregunta tan temida en Análisis Matemático: ¿qué es una función? 😄 Es la relación entre un dominio e imagen, en donde, para una entrada tenemos una salida (existencia) y esa salida es única (unicidad). Esta misma norma se va a cumplir para las *funciones* que creemos en `Haskell`, el lenguaje correspondiente a este paradigma.

Dicho esto, adentrémonos en Haskell. Estos son los ejemplos de funciones que vimos en clase con sus respectivos tipos:

```
doble :: Int -> Int
doble numero = numero * 2

siguiente :: Int -> Int
siguiente numero = numero + 1

sumaDe4Numeros :: Int -> Int -> Int
sumaDe4Numeros primerNumero segundoNumero tercerNumero cuartoNumero = primerNumero + se
```

Es importante tener en cuenta que el tipo de una función NO es el tipo de su retorno, sino que está compuesto por el tipo de sus valores de entrada y el de salida. Recordemos como regla mnemotécnica que la cantidad de flechas del tipo es igual a la cantidad de parámetros que tiene la función.

Ahora supongamos que queremos ver si queremos saber el doble del siguiente de un número. ¿Cómo lo resolvemos? 😱 Usando **composición**:

```
> (doble.siguiente) 2
=> 6
```

¿Qué lo qué está pasando? 😱 Lo mismo que en la composición de funciones matemáticas (FoG(x)) . Primero se aplica la función de la derecha con el valor y luego se aplica la de la izquierda con el valor que nos devolvió la función anterior.

Recordemos que, como en matemática, el valor que retorne la función de la derecha, tiene que ser un valor que la función de la izquierda pueda operar. Es decir, que la imagen de la función de la derecha esté incluida en el dominio de la función de la izquierda.

Y ya que estamos, démosle un nombre al cálculo del doble del siguiente. Lo haremos creando una nueva función llamada... ¡`dobleDelSiguiente`!

```
dobleDelSiguiente unNumero = (doble.siguiente) unNumero
```

Y entonces, lo que nos queda después de componer dos funciones es... ¡una nueva función! 😮

Inmutabilidad

Estos son los ejemplos de funciones y valores que vimos en clase:

```
frecuenciaCardiacaPromedio = 80

hacerActividadFisica unaFrecuencia = unaFrecuencia + 50

tieneTaquicardia unaFrecuencia = unaFrecuencia >= 180
```

De esa forma le pusimos un alias o etiqueta al valor 80 con `frecuenciaCardiacaPromedio` y creamos funciones como `hacerActividadFisica` y `tieneTaquicardia`. Las funciones van a ser nuestras herramientas para poder operar a los valores.

Algo muy importante es que en Haskell **no hay efecto**. Esto quiere decir que los valores igualados no van a mutar luego de ser operados por las funciones. Este concepto se llama **inmutabilidad**.

Por ejemplo, si aplicamos `hacerActividadFisica` a la `frecuenciaCardiacaPromedio`, podemos ver que `frecuenciaCardiacaPromedio` no cambia su valor:

```
> frecuenciaCardiacaPromedio
=> 80
> hacerActividadFisica frecuenciaCardiacaPromedio
=> 130
> frecuenciaCardiacaPromedio
=> 80
```

Por esto, es que en Haskell logramos tener lo que se llama **transparencia referencial**. Es importante recordar que `frecuenciaCardiacaPromedio` no es una variable, sino que es simplemente un alias, es decir, otra manera de decirle al valor 80.

Composición

Ahora supongamos que queremos ver si tenemos taquicardia después de hacer actividad física. ¿Cómo lo resolvemos? 😱 Usando **composición**:

```
> (tieneTaquicardia.hacerActividadFisica) 70
=> True
```

Como ya mencionamos anteriormente, el valor que retorne la función de la derecha, tiene que ser un valor que la función de la izquierda pueda operar. Si quisiéramos componerlo al revés:

```
> (hacerActividadFisica.tieneTaquicardia) 70
```

Va a romper ya que `hacerActividadFisica` tiene que recibir un número, y está recibiendo un booleano.

Por último, vamos a darle un nombre a la acción de preguntar si se tiene taquicardia luego de hacer una actividad física creando la función`tieneTaquicardiaDespuesDeEntrenar`:

```
tieneTaquicardiaDespuesDeEntrenar unaFrecuencia = (tieneTaquicardia.hacerActividadFisic
```

Que no exista el estado en Haskell, hace que la composición tenga más relevancia. Ya que como no podemos pisar valores con variables, la composición nos permite encadenar las funciones para trabajar con diferentes valores y así poder crear soluciones más complejas. ✨

Precedencia de operadores

En matemática, cuando tenemos una expresión como 2 * 3 + 4, solemos operarla dependiendo de la precedencia de cada operador. Como el * es de mayor precedencia que el +, operamos primero 2 * 3 y luego le sumamos 4.

En Haskell también se respeta esto. Les dejamos una tabla para que puedan ver la precedencia que utiliza Haskell:

Precedencia (Mayor número, mayor precedencia)	“Operador”
11	()
10	Aplicacion prefija
9	.
8	^
7	*,/
6	+,-
5	:
4	==, /=, <, <=, >, >=
3	&&
2	||
1	$

Type classes

Pensemos en la función `suma`:

```
suma unNumero otroNumero = unNumero + otroNumero
```

¿Qué tipo debería tener?

¿Enteros?

¿Que tal `suma :: Int -> Int -> Int`?

Dados estos números:

```
unEntero :: Int
unEntero = 2

otroEntero :: Int
otroEntero = 3

unFlotante :: Float
unFlotante = 2
```

```
otroFlotante :: Float
otroFlotante = 3
```

¿Qué creen que pasaría si queremos evaluar la siguiente expresión: `suma unEntero otroEntero`?

```
> suma unEntero otroEntero
5
```

¿y `suma unFlotante otroFlotante`?

```
> suma unFlotante otroFlotante
-- * estalla *
```

¡Tiene sentido! le dijimos a nuestra función suma que su dominio son los enteros, entonces cuando le damos un flotante, nos dice “eh, no; yo trabajo sólo con enteros”

¿Flotantes?

¿Y qué tal si la hubiésemos definido como `suma :: Float -> Float -> Float`? Después de todo, los enteros son un subconjunto de los reales, ¿no?

```
> suma unFlotante otroFlotante
5
```

```
> suma unEntero otroEntero
-- * estalla *
```

Bueno, no; si bien en la matemática es cierto que los enteros son reales, en definitiva para haskell `Float` e `Int` son tipos de datos distintos.

¿a?

¿Y si hago `suma :: a -> a -> a`?

Falla al cargar el archivo

Parece que haskell no nos permite sumar cualquier cosa tampoco, lo cual es de esperarse; ¿tendría sentido que nos deje sumar dos funciones? ¡no!

¿Pero qué onda? Si yo en la consola hago...

```
> unFlotante + otroFlotante
5
```

ó

```
> unEntero + otroEntero
5
```

¡Me andan las dos!

¡Números!

Para expresar el tipo de `suma`, en realidad nos está faltando una herramienta, a la cual llamamos **Familia de Tipos**, o **Type Class**.

Mientras que a un tipo lo podríamos describir como un conjunto de valores asociado a un conjunto de operaciones con las que los podemos trabajar, una

familia de tipos es más bien un contrato que te dice qué operaciones tiene que entender un tipo para pertenecer a esa familia.

Hasta acá suena todo muy abstracto, así que bajémoslo a un ejemplo concreto: ¡los números!

Esa cosa en común que tienen `Int` y `Float` que nos permite sumarlos es la familia de tipos de los números `Num`; y su contrato nos dice que cualquier tipo que pertenezca a su familia, se puede sumar `(+)`, restar `(-)`, y multiplicar `(*)`!

¿Y esto cómo lo escribimos en el código?

```
suma :: Num a => a -> a -> a
suma unNumero otroNumero = unNumero + otroNumero
```

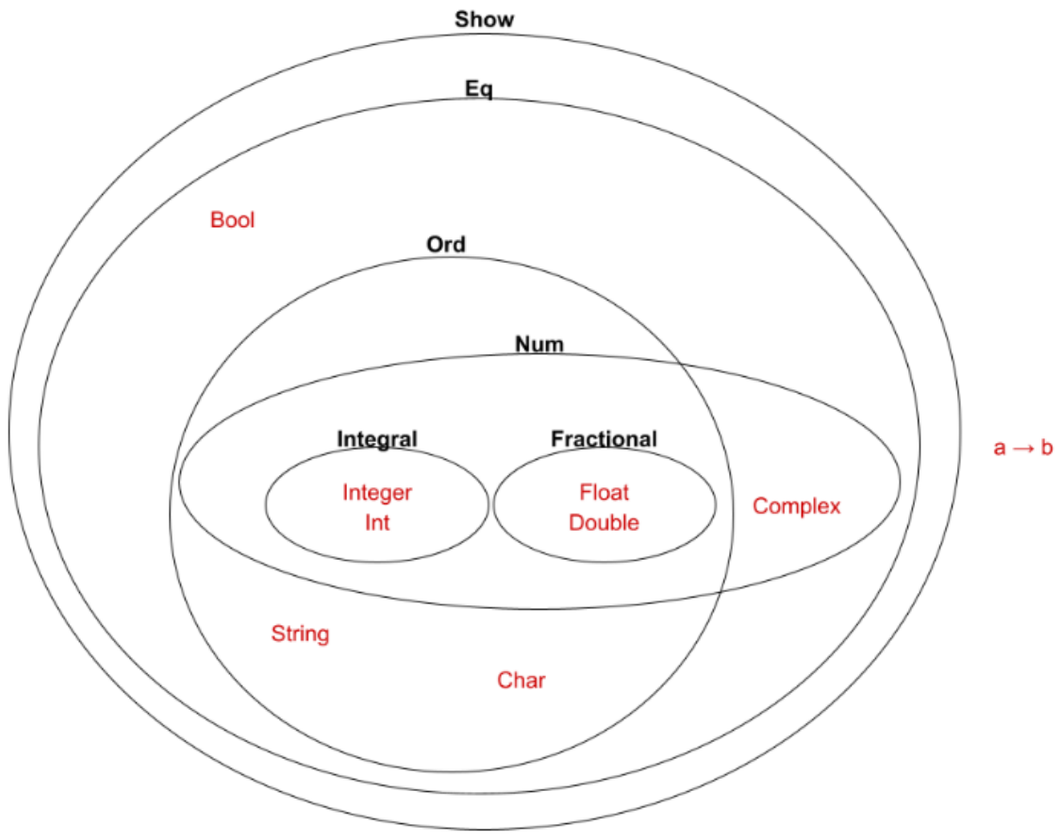
Cuando escribimos esto, estamos restringiendo a que el tipo `a` debe pertenecer a la familia de tipos `Num`.

Es importante recalcar que en toda la firma de `suma`, `a` representa a un **mismo tipo**. Si bien ahora podemos sumar enteros con enteros, y flotantes con flotantes, esto **no** nos permite sumar enteros con flotantes.

Y así como tenemos una familia de tipos para los números, tenemos otro montón con distintos propósitos, como:

- Show: Las cosas que se saben mostrar por pantalla (en la consola).
- Eq: Las cosas que se saben comparar por igualdad `(==)`.
- Ord: Las cosas que se saben comparar por orden `(>)`, `(<)`, etc.
- Num: ¡Los números! se saben sumar, restar y multiplicar.
- Integral: Para números enteros; entienden la división entera (`div`), el resto de la división `rem`, se les puede preguntar si son pares (`even`) o impares (`odd`).
- Fractional: Para números reales, que se pueden dividir con la división flotante `(/)`

Podemos los tipos que pertenecen a cada familia en el siguiente diagrama:



Es posible que haya alguna mentira blanca en este diagrama

Links Útiles

- Concepto de función
- Composición
- Tipos en Haskell
- Inferencia de tipos

Podes ver nuestro github acá.

Y saber mas sobre nosotros acá

← Presentación y primeros conceptos

Aplicación parcial y orden superior →