Facultad de Ciencias - UNAM
Estructuras Discretas 2023-1
Práctica 6: Clases de tipos

Laura Friedberg Gojman
Saúl Adrián Rojas Reyes
José Manuel Madrigal Ramírez

24/marzo/2023
Fecha de entrega: 2/abril/2023 a las 23:59 hrs

Instrucciones

Una vez más, junto a este PDF, he subido un archivo llamado `Practica6.hs`. Descárgalo y ábrelo en tu editor de texto preferido. Allí encontrarás instrucciones que se corresponden con los ejercicios mencionados en esta práctica. Abre también una terminal situada en la carpeta donde guardaste el archivo.

La dinámica de trabajo será similar a la que seguimos en las prácticas anteriores, y me parece que así será hasta el fin del curso. Cualquier sugerencia que tengas para mejorar esta dinámica puedes incluirla en los comentarios del archivo *ReadMe.txt* que acompañe la entrega de tu práctica.

Introducción

Entrando en materia, podemos pensar a los tipos de dato (`Int, Double, String`, etc.) como conjuntos que contienen variables con ciertas propiedades parecidas. Así tenemos al conjunto de los enteros (*Int*), al conjunto de los carácteres (*Char*) etc. Siguiendo este orden de ideas, el conjunto al que pertenecen ciertos tipos de dato similares entre sí sería un conjunto de conjuntos. Estaríamos hablando de una ***clase***. En esta práctica abordaremos las *clases de tipos* y otros conceptos que nos ayudarán a escribir funciones de propósito más general.

Recordemos que Haskell tiene un *sistema de tipos estático*, esto quiere decir que su compilador detecta todos los errores de correspondencia entre tipos. Por ejemplo, si realizamos una operación que está definida para dos enteros (como `div`, que era la división para Int) y le pasamos un par de datos de tipo Double,

el compilador nos enviará un error. En otros lenguajes esto no suele suceder pues hay conversiones implícitas de tipo. Pese a que en primera instancia esto parezca una desventaja, en realidad es una propiedad que vuelve a Haskell un lenguaje mas fiable.

Para facilitarnos las cosas Haskell también cuenta con un *sistema de inferencia de tipos* que podemos utilizar desde GHCi con el comando `:t`. En la parte superior del archivo `practica6.hs` hay cuatro variables definidas, veamos cómo utilizar el comando `:t` para inferir el tipo de dos de ellas:

```
*Practica6>:t x
x :: Int
*Practica6>:t y
y :: Int
```

Ejercicios

1. Utiliza el comando `:t` en consola para inferir el tipo de las variables `z` y `f` que se encuentran definidas en el archivo `practica6.hs`. Saca captura de pantalla de tus resultados.

2. Podemos ver que el tipo de la función `head` es:

   ```
   [a] -> a
   ```

   El símbolo `a` representa una *variable de tipo*. De modo que esta función puede aceptar listas de cualquier tipo siempre que el dato de salida sea de ese mismo tipo. Por ejemplo, si utilizamos una lista de `Int` como entrada, por fuerza el dato de salida debe ser también de tipo `Int`.

   Decimos que `head` y todas las funciones que se definan utilizando *variables de tipo* son llamadas **funciones polimórficas**.

   Definamos una función polimórfica llamada `snocP` que coloque un elemento al final de una lista dada. Aquí algunos ejemplos de su funcionamiento:

   ```
   *Practica6>snocP ['a','b','c'] '='
   "abc="
   *Practica6>snocP ['a','b','c'] 'k'
   "abck"
   *Practica6>snocP [[1],[2],[10]] [2,3,4]
   [[1],[2],[10],[2,3,4]]
   ```

3. El comando `:t` también nos sirve para exhibir el tipo de los operadores, para ello debemos colocarlos entre paréntesis. Veamos un ejemplo de esto con el operador `+`:

   ```
   *Practica6>:t (+)
   (+) :: Num a =>a ->a ->a
   ```

   Utiliza el comando `:t` en GHCi para inferir el tipo de los siguientes operadores y guarda capturas de tus resultados:

- ==
- >=
- *

En realidad, todos los operadores son funciones. Cuando el nombre de una función está compuesto por carácteres especiales (no alfanuméricos), se considera una función *infija* por defecto. Si queremos examinar su tipo, pasarla a otra función o llamarla en forma prefija debemos rodearla con paréntesis. Por ejemplo: (+) 1 4 equivale a 1 + 4.

4. Observa que en los resultados del ejercicio anterior el tipo de cada operador incluía, además de las variables de tipo, las palabras `Num`, `Ord` y `Eq` y el símbolo `=>`.

   `Num`, `Ord` y `Eq` son ejemplos de **clases de tipos**. Para pertenecer a una determinada clase, un tipo debe cumplir ciertas propiedades. Por ejemplo, los tipos Int, Float y Double pertenecen a la clase `Num` porque sus miembros pueden comportarse como números.

   Detrás del símbolo `=>` colocamos las *restricciones de clase*. Esto quiere decir que los tipos de los valores señalados deben pertenecer a la clase indicada. Por ejemplo, con el operador `+` obtuvimos: `Num a =>a -> a -> a`. Y esto indica que los tipos a los que pertenezcan los valores con los que usemos la suma, deben pertenecer a la clase *Num*, es decir, deben comportarse como números.
   Por eso es que podemos usar el operador de suma con valores de tipo `Float, Int o Double`. Esos tres tipos, como dijimos anteriormente, son de la clase `Num`.

   Investiga y escribe con tus propias palabras en un archivo ReadMe.txt, las propiedades de las siguientes clases de tipo:

   - `Show`
   - `Enum`
   - `Bounded`
   - `Read`

5. Utiliza guardas para definir una función `n_abs` que regrese el valor absoluto de un número entero o decimal:

```
*Practica6>n_abs 7.2
7.2
*Practica6>n_abs (-8.5)
8.5
*Practica6>n_abs (-2)
2
```

6. Define una función `listasIguales` que recibe dos listas y nos indica si sus elementos son iguales.

```
*Practica6>listasIguales [1,2,3,4,5] [1..5]
True
*Practica6>listasIguales ‘‘Hola’’ ‘‘Hola’’
True
*Practica6>listasIguales [1..5] [1,5,6,4,5]
False
```

7. Define una función *n_ elem* que nos indique si un elemento dado pertenece a una lista.

```
*Practica6>n_elem 'h' ‘‘holaaaaa’’
True
*Practica6>n_elem 3 [1..4]
True
*Practica6>n_elem 3.7 [1.4, 3.7]
True
*Practica6>n_elem 'h' ‘‘Holaaaaa’’
False
```

8. Define una función *inserta* que, dada una lista ordenada de menor a mayor, coloca un elemento en su lugar, preservando el orden de la lista.

```
*Practica6>inserta 'h' ‘‘opq’’
‘‘hopq’’
*Practica6>inserta 1 [-1, 4, 100]
[-1,1,4,100]
*Practica6>inserta 3.5 [1,2,8]
[1.0,2.0,3.5,8.0]
```

9. Recordemos de la práctica anterior la definición recursiva que dimos de un árbol binario:

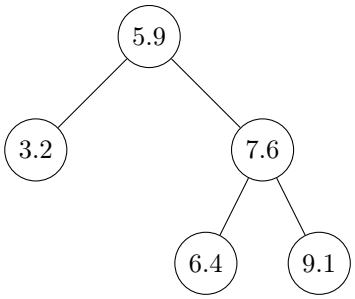
```haskell
data Arbol =  Vacio
            |  Nodo Arbol Int Arbol
      deriving (Show, Eq)
```

Podemos redefinirlo para que sea un árbol binario de cualquier tipo de dato añadiendo una *variable de tipo* `a`:

```haskell
data Arbol a =  Vacio
              |  Nodo (Arbol a) Int (Arbol a)
      deriving (Show, Eq)
```

Veamos algunos ejemplos de árboles binarios definidos con la modificación que hicimos para aceptar datos de cualquier tipo en los nodos:

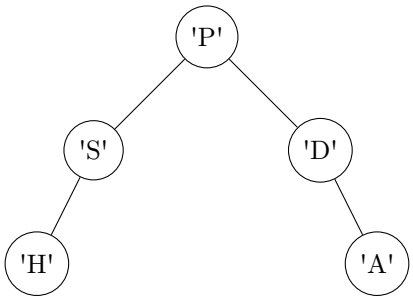
Árbol 1:



Código 1:

```haskell
ej1 :: Arbol Double
ej1 =  Nodo ( Nodo  Vacio 3.2  Vacio)
       5.9
       ( Nodo ( Nodo  Vacio 6.4  Vacio)
       7.6
       ( Nodo  Vacio 9.1  Vacio))
```

Árbol 2:



Código 2:

```haskell
ej2 :: Arbol Char
ej2 =  Nodo ( Nodo ( Nodo  Vacio 'H'  Vacio) 'S'  Vacio)
       'P'
       ( Nodo  Vacio 'D' ( Nodo  Vacio 'A'  Vacio))
```

Define una función **numHojas** que cuente el número de hojas de un árbol dado:

```
*Practica6>numHojas ej1
3
*Practica6>numHojas ej2
2
```

Pista: Define una función auxiliar que indique si un nodo es una hoja.

Entrega

La entrega es **individual** y consistirá en un archivo comprimido que contenga:

- El archivo `Practica6.hs` con el código indicado en los ejercicios.
- Las capturas de pantalla que piden los ejercicios 1 y 3.
- Un archivo ReadMe.txt que incluya tu nombre, la respuesta al ejercicio 4 y algún comentario o idea sobre la práctica. Opiniones y sugerencias también son bien recibidos.

Sube el archivo comprimido a la plataforma Google Classroom utilizando el siguiente formato para el nombre:

Practica06_ Apellido1-Apellido2-Nombre(s).zip

Cualquier duda que tengas no dudes en enviarme un correo. 😄

Enlace de apoyo

- Creando nuestros propios tipos y clases de tipos, Aprende Haskell por el bien de todos