Facultad de Ciencias - UNAM
Estructuras Discretas 2023-1
Práctica 4: Tipos de dato en Haskell

Laura Friedberg Gojman
Saúl Adrián Rojas Reyes
José Manuel Madrigal Ramírez

3/marzo/2023
Fecha de entrega: 12/marzo/2023

Instrucciones

Junto a este PDF, subí un archivo llamado `Practica3.hs`. Descárgalo y ábrelo en tu editor de texto preferido. 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. En esta ocasión veremos cómo crear nuestros propios tipos de dato y cómo definir funciones con ellos.

Ejercicios

Tipos enumerados

1. Los tipos de dato más básicos que podemos crear en Haskell son los tipos de dato *enumerados*. Reciben ese nombre porque podemos *enumerar* todos los valores posibles que pueden tomar. A continuación tenemos la definición del tipo de dato `Cosa`. *Cópiala en el archivo de la práctica* (no vale usar ctrl+c y ctrl+v, procura escribirla por tu cuenta):

```haskell
data  Cosa = Soda
           | Barco
           | Jabon
           | Repollo
           | Rascacielos
           | Taquito
```

   Observa que la palabra clave *data* es la que nos permite declarar nuestros propios tipos de dato.

2. A las variables del tipo `Cosa` se les puede asignar uno de los seis posibles valores presentes en la definición, veamos:

```
x :: Cosa           -- x es del tipo Cosa
x = Repollo         -- definimos a x con el valor 'Repollo'
```

Escribe otras tres variables del tipo de dato `Cosa`.

3. Al abrir una terminal, cargar el código de la práctica en `ghci` y querer ver el valor de alguna de las variables que acabamos de declarar obtendremos un error.

Saca captura de pantalla de éste error y guárdala.

Si escribimos *deriving* `Show` debajo de nuestra definición podremos ver el valor de las variables en pantalla:

```
data Cosa = Soda
          | Barco
          | Jabon
          | Repollo
          | Rascacielos
          | Taquito
       deriving Show
```

Crea un archivo `readMe.txt` *investiga y responde ¿para qué sirve la palabra reservada* `deriving`?

4. *Define una función*

`esGrande :: Cosa ->Bool`

que devuelve `True` *si el objeto es grande en relación a una persona y* `False` *en caso contrario.* Aquí algunos ejemplos de su funcionamiento en terminal:

```
*Practica3>esGrande Barco
True
*Practica3>esGrande Jabon
False
```

5. También podemos tener sinónimos de tipos de dato. Esto se consigue utilizando la palabra reservada *type*. *Copia la siguiente línea de código en el archivo de la práctica:*

```
type Alcaldia = String
```

Con lo anterior podemos crear variables del tipo `Alcaldia` *Cópialas en el archivo o crea tus propias variables*:

```
iz , cy , tl :: Alcaldia
iz = "Iztapalapa"
cy = "Coyoacán"
tl = "Tlahuac"
```

En el fondo sabemos que su verdadero tipo es `String`, pero al crear sinónimos de tipo ganamos legibilidad en nuestro código.

Define un sinónimo para el tipo `Int` *que se llame* `Edad`.

Tipos de dato algebraicos

Los tipos algebraicos tienen **argumentos** que nos permiten añadir información. Al definir un tipo de dato algebraico utilizamos **constructores** que especifican el tipo de dato de dichos argumentos.

1. *Copia la siguiente declaración del tipo* `Persona` *cuyo único constructor recibe cuatro argumentos:*

```
data Persona = Persona String  Edad  Cosa  Alcaldia
    deriving Show
```

   Observa como el nombre del tipo de dato `Persona` y el constructor `Persona` reciben el mismo nombre pero significan cosas distintas.

   Veamos un ejemplo de cómo definir variables con este nuevo tipo de dato:

```
k ::  Persona              --aquí Persona es un tipo de dato
k =  Persona "Kath" 19 Soda  cy      --aquí es un constructor
```

   Define tres variables de tipo persona.

2. Ahora nos toca definir funciones que devuelvan cada uno de los atributos de una variable del tipo `Persona`:

   - Nombre
   - Edad
   - Cosa
   - Alcaldía

   Aquí, por ejemplo, tenemos la función que devuelve la edad:

```
edad ::  Persona ->  Edad
 edad ( Persona _ a _ _) = a
```

   Veamos su ejecución en terminal con la variable que definimos antes:

```
*Practica3>edad k
19
```

3. Aquí tenemos la sintaxis general para construir tipos de dato algebraicos:

```
data TipoDeDatoAlgebraico = Constructor1 Tipo11 Tipo12
                          | Constructor2
                          | Constructor3 Tipo31 Tipo32 Tipo33
                          | Constructor4 Tipo41
```

Podemos tener constructores sin parámetros como el `Constructor2`; con un parámetro como el `Constructor4` o con muchos parámetros como el `Constructor1` o el `Constructor3`. Observa el protagonismo del símbolo "|" para separar distintos constructores.

Añade un constructor llamado `Estudiante` *al tipo de dato* `Persona` *con un atributo extra para el número de cuenta. Cuida el tipo de dato que elijas para este atributo.*

4. *Finalmente, crea dos variables de tipo* `Estudiante` *y una función que devuelva el número de cuenta.*

   Veamos unos ejemplos de su ejecución en terminal (asumiendo que `a` y `b` son variables del tipo `Persona` creadas con el constructor `Estudiante` y cuyos números de cuenta son `420077875` y `319274830` respectivamente:

```
*Practica3>numeroCuenta a
420077875
*Practica3>numeroCuenta b
319274830
```

   Como una observación final, respecto a la sintaxis, notemos que los nombres de los **tipos de dato** siempre empiezan con **mayúscula**, mientras que las **variables** (incluyendo los nombres de funciones) siempre se n empezando con **minúscula**.

Correspondencia de patrones

La *Correspondencia de patrones* se refiere a **seleccionar un valor** dentro de una función **dependiendo del constructor** con el que fue **definido**.
Por ejemplo, observemos la siguiente versión de la función `edad`:

```
1  edad :: Persona -> Edad
2  edad ( Persona _ a _ _) = a
3  edad ( Estudiante _ a _ _ _) = a
```

En esta versión la función `edad` puede ejecutarse adecuadamente tanto con variables que se hayan definido utilizando el constructor `Persona` como con el constructor `Estudiante`.

1. *Modifica las funciones* `nombre`, `edad`, `cosaFav` *y* `alcaldia` *para que sirvan también con variables que se hayan definido utilizando el constructor* `Estudiante`.

2. *Modifica la función* `numeroCuenta` *para que regrese un* `0` *en caso de que la persona en cuestión no sea estudiante.*

```
*Practica3>numeroCuenta k
0
*Practica3>numeroCuenta b
319274830
```

3. *Copia las siguiente función en el archivo de la práctica y modifícala para que funcione adecuadamente con variables que se hayan definido utilizando el constructor* `Estudiante`:

```
1   presentacion ::  Persona -> String
2   presentacion p@( Persona _ _ _ _) = "La persona " ++
3             (nombre p)  ++ " vive en " ++ (alcaldia p)
```

   El símbolo `@` presente en la linea `2` sirve para comparar el valor `p` con el patrón `(Persona n _ _ _ )`. En caso de haber correspondencia, se le asigna el identificador p al valor de entrada. De esta forma podemos trabajar con él en la definición de la función. Observa cómo lo utilizamos para llamar a las funciones `nombre` y `alcaldia`.

4. *Copia la siguiente función en el archivo de la práctica y modifícala añadiendo otro par de cláusulas para variables que se hayan definido utilizando el constructor* `Estudiante`.

```
1 favorito ::  Persona -> String
2 favorito ( Persona n _ Taquito _) = n ++ ", vamos por tacos UuU"
3 favorito ( Persona n _ Soda _)    = n ++ ", mejor toma agua x("
```

   Observa que los patrones pueden anidarse. En este caso estamos anidando el patrón correspondiente a los constructores del tipo `Cosa`, dentro de los patrones correspondientes a los constructores del tipo `Persona`.

5. *Finalmente añade otras frases provenientes de tu ingenio para el resto de objetos del tipo* `Cosa` *o bien, un caso genérico para el resto de cosas con una sola frase. Procura que funcione con personas y estudiantes.*

Aquí tenemos una gramática que define a los patrones:

```
pat ::= _
    | var
    | var @ ( pat)
    | (Constructor pat1 pat2 ... patn)
```

Es decir:

- Un guión bajo es un patrón (suele utilizarse cuando esa parte del dato de entrada no nos importa)
- Una variable es un patrón
- Una variable que encaja con un patrón, es a su vez un patrón
- El identificador de un constructor seguido de patrones es, a su vez, un patrón

A lo largo de esta práctica hemos visto ejemplos de cada uno ¿Puedes identificarlos?

Entrega

Esta vez la entrega será **individual** y consistirá en un archivo comprimido que contenga:

- El archivo `Practica4.hs` con el código indicado en los ejercicios.
- La captura de pantalla del error en el ejercicio 3.
- Un archivo readMe.text que contenga
  - La respuesta a la pregunta del ejercicio 3
  - Comentarios, opiniones y sugerencias. (Son muy importantes, me dan retroalimentación) 

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

Practica04_ Apellido1-Apellido2-Nombre(s).zip

Algunos enlaces de apoyo

- Introducción a Haskell por Brent Yorgey de la Universidad de Pensilvania
- Tipos de datos en Haskell, por la Facultad de Matemáticas, Astronomía y Física
- Creando nuestros propios tipos y clases de tipos, Aprende Haskell por el bien de todos

Cualquier duda que tengan recuerden que pueden en enviarme un correo o comunicarse por telegram.