Lógica Computacional, 2019-2
Nota de laboratorio 3: Definición de tipos de datos

Manuel Soto Romero

20 de febrero de 2019

Facultad de Ciencias, UNAM

Adicional a los tipos de datos primitivos, Haskell provee algunas primitivas para definir tipos de datos que no están incluidos en el núcleo del lenguaje. En esta nota se revisan dichas primitivas y se revisan algunos ejemplos de funciones que operan sobre estos tipos.

1. Sinónimos

Una forma de definir tipos es mediante lo que se conoce como *sinónimo*. Un sinónimo permite renombrar tipos de datos de Haskell de manera que sea más claro para qué se están utilizando dentro del programa.

Un claro ejemplo de sinónimo son las cadenas, en la Nota de laboratorio 2 se mencionó que una cadena es simplemente una lista de caracteres, de esta forma, se usa un sinónimo para renombrar a las listas de este tipo definiendo así el tipo `String` que tiene exactamente el mismo comportamiento que el tipo original.

Ejemplo 1. Sinónimo para cadenas.

```haskell
type String = [Char]
```

Listado de código 1: Sinónimo de listas de caracteres

□

Ejemplo 2. Para la biblioteca de vectores definida en la Nota de laboratorio 2, puede definirse un sinónimo vector como se aprecia en la línea 4 del Listado de código 2. La firmas de las funciones se modifican como se aprecia en las líneas 7, 11 y 15. Esta forma de definir funciones permite entender al programador que se usan tuplas para representar vectores.

```haskell
module Vectores where

   -- Definición del tipo vector mediante un sinónimo.
   type Vector = (Float,Float)

   -- Función que suma dos vectores.
   suma :: Vector -> Vector -> Vector
   suma u v = (fst u + fst v, snd u + snd v)

   -- Función que realiza el producto punto de dos vectores.
   productoPunto :: Vector -> Vector -> Float
```

```
12    productoPunto u v = fst u * fst v + snd u * snd v
13
14    -- Función que realiza el producto de un vector y un escalar.
15    productoEscalar :: Vector -> Float -> Vector
16    productoEscalar u k = (fst u * k, snd u * k)
```

Listado de código 2: Biblioteca para trabajar con vectores

□

También es posible definir sinónimos parametrizados, esto quiere decir, que su definición especifica una variable de tipo que puede ser usada durante la construcción del tipo en tiempo de ejecución.

Ejemplo 3. Definición de pares mediante sinónimos parametrizados. De esta forma pueden definirse pares genéricos. La función de las líneas 7 a 8 usa pares con los tipos numéricos de la clase `Num`, mientras que la función de las líneas 11 a 12 usa pares de cualquier tipo, es decir, es una función polimórfica.

```
1  module Pares where
2
3     -- Definición del tipo par mediante un sinónimo.
4     type Par a = (a,a)
5
6     -- Multiplicación de los elementos de un par.
7     multiplica :: Num a => Par a -> a
8     multiplica (x,y) = x*y
9
10    -- Función que copia el elemento recibido en un par.
11    copia :: a -> Par a
12    copia x = (x,x)
```

Listado de código 3: Pares genéricos

□

2. Definición de tipos de datos

Para definir tipos de datos nuevos, Haskell provee la primitiva `data`.

Ejemplo 4. Definición del tipo `Arbol` para representar árboles binarios.

```
1  data Arbol a = Hoja a
2               | Nodo (Arbol a) a (Arbol a)
3               deriving Show
```

Listado de código 4: Definición de árboles binarios

□

Del Listado de código 4 se puede apreciar:

- Se define el nuevo tipo de dato `Arbol` mediante la primitiva `data`.
- El identificador `Arbol` a la derecha del símbolo de igualdad (=) se conoce como *constructor de tipo*. En este caso el constructor de tipo no recibe parámetros.
- Los identificadores `Hoja` y `Nodo` son llamados *constructores de datos* que en este caso reciben algunos tipos como parámetro que indican el tipo de dato que almacenan. En este caso el tipo `Arbol` es recursivo pues el constructor de datos `Nodo` recibe parámetros de tipo `Arbol`.
- Los constructores de tipo y de datos no necesariamente deben recibir parámetros. El tipo de los parámetros puede ser cualquier tipo concreto o definirlo polimórficamente mediante una variable de tipo.
- El símbolo de barra vertical o *pipe* (|) se lee como "o" y permite separar los distintos constructores de datos. *El tipo de dato árbol es una hoja o un nodo*.
- Los constructores de datos son en realidad funciones que reciben los tipos especificados en los parámetros y regresan algo del tipo al que pertenecen.

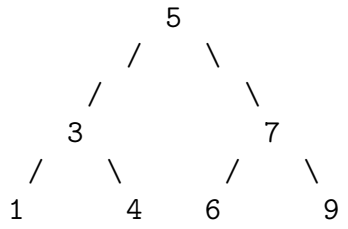
  ```
  Hoja :: a -> Arbol a
  Nodo :: Arbol a -> a -> Arbol a -> Arbol a
  ```

- Para poder mostrar un árbol desde terminal, éste debe formar parte de la clase de tipos `Show`. Para hacer que un nuevo tipo de dato forme parte (derive) de una clase de esta, se usa la primitiva `deriving` indicando clase a la que deriva.

  Observación 1. Para que una función forme parte de una clase, ésta debe implementar las funciones requeridas por la misma. Si un tipo deriva de una clase, la implementación que se toma es la implementación por defecto de la clase, si es que ésta cuenta con una.

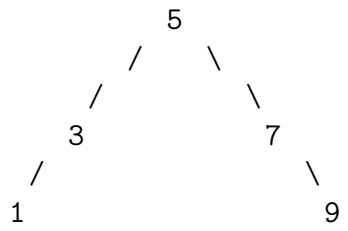
  La clase `Show` requiere que se implemente la función `show` e incluye una implementación por defecto.

- La siguiente ejecución muestra un árbol con 7 elementos de tipo entero:




```
Arboles> Nodo (Nodo (Hoja 1) 3 (Hoja 4)) 5 (Nodo (Hoja 6) 7 (Hoja 9))
Nodo (Nodo (Hoja 1) 3 (Hoja 4)) 5 (Nodo (Hoja 6) 7 (Hoja 9))
```

¿Es posible construir el siguiente árbol usando esta definición de árboles? ¿Por qué?



A continuación se definen algunas funciones para trabajar con el tipo `Arbol`. Al crear un tipo de dato mediante data, es posible usar la técnica de reconocimiento de patrones a través de los constructores de datos.

Ejemplo 5. Definir la función `contiene` tal que `(contiene e a)` indica si el elemento e se encuentra en el árbol a.

```haskell
-- Función que indica si un elemento se encuentra contenido en el
-- árbol.
contiene :: Eq a => a -> Arbol a -> Bool
contiene e (Hoja x)     = e == x
contiene e (Nodo i x d) = e == x || contiene e i || contiene e d
```

Listado de código 5: Función contiene sobre árboles binarios

□

Ejemplo 6. Definir la función `aplana` tal que `(aplana a)` es la lista obtenida a partir de los elementos del árbol a.

```haskell
-- Función que aplana un árbol binario.
aplana :: Arbol a -> [a]
aplana (Hoja x)     = [x]
aplana (Nodo i x d) = aplana i ++ [x] ++ aplana d
```

Listado de código 6: Función aplana sobre árboles binarios

□