Facultad de Ciencias - UNAM
Estructuras Discretas 2023-1
Práctica 5: Tipos de dato recursivos

Laura Friedberg Gojman
Saúl Adrián Rojas Reyes
José Manuel Madrigal Ramírez

10/marzo/2023
Fecha de entrega: 19/marzo/2023

Instrucciones

Junto a este PDF, subí un archivo llamado `Practica5.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. Vamos a extender nuestros conocimientos sobre la declaración de tipos de dato, en la práctica pasada abordamos los tipos de datos enumerables y algebraicos, ahora nos estudiaremos los tipos de dato recursivos.

Breve introducción

De manera semejante a como definimos funciones utilizando recursión, podemos definir ciertos conjuntos infinitos como los números naturales, las listas, los árboles binarios o las expresiones lógicas.

En general, la característica que debe tener un conjunto para que lo podamos definir con recursión es que sea **numerable**, es decir, que dado un elemento podamos determinar cual es el siguiente. Por ejemplo, pensemos en un número natural cualquiera, digamos el 3. Es claro que el elemento siguiente es el 4.

Por el contrario pensemos en el 3 dentro de los **números reales** $\mathbb{R}$. Podemos ver que 3.1 no es el siguiente elemento, pero tampoco lo es 3.01, ni 3.001 etc. Sin lugar a dudas es imposible determinar el siguiente elemento después del 3. Así que los números reales son un conjunto infinito que no se puede contar, esto es, los números reales son **no numerables** en el sentido de que dado un elemento del conjunto, no podemos determinar el elemento siguiente.

Definamos los números naturales de manera recursiva. Para ello utilizaremos una codificación que únicamente empleará la función sucesor $suc(n) = n + 1$ y el 0:

0	cero
1	suc(cero)
2	suc(suc (cero))
3	suc(suc (suc (cero)))
4	suc(suc(suc (suc (cero)))
.	.
.	.
.	.

En Haskell tendríamos:

```haskell
data Nat = Cero
         | Suc  Nat
    deriving (Eq, Show)
```

Donde:

- *Nat*: es el tipo de dato.
- `Cero`: es un constructor sin parámetros
- `Suc`: es un constructor que recibe un parámetro de tipo *Nat* (de aquí que es una definición recursiva).
- `Show`: Permite que nuestro tipo de dato se muestre en pantalla.
- `Eq`: Permite que nuestro tipo de dato sea comparable con el símbolo `==`

Con este tipo de dato podemos declarar nuestras variables del tipo *Nat* en algún documento `.hs`:

```haskell
uno ::  Nat
uno = Suc Cero

dos ::  Nat
dos = Suc (Suc Cero)
```

O en GHCi de la siguiente manera:

```
*Main>z = Cero
*Main>z
Cero
*Main>tres = Suc (Suc (Suc Cero))
*Main>tres
Suc (Suc (Suc Cero))
```

Ejercicios

1. Transcribe en el archivo `Practica5.hs` la definición del tipo de dato *Nat* y declara variables para el ocho, el cuatro y el dos.

2. *Define una función* `aNatural :: Int ->Nat` *que convierta un entero positivo en un natural como los que acabamos de definir. En caso de que la entrada sea un número negativo devolveremos* `Cero`.

   Veamos algunos ejemplos de su ejecución:

```
*Main>aNatural 10
Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc Cero)))))))))
*Main>aNatural (-2)
Cero
*Main>aNatural 0
Cero
```

   Pista: Podemos usar guardas para procesar el caso en que la entrada sea un número negativo. Por otra parte para el caso recursivo, podemos pensar que aplicar `aNatural` a un número n es lo mismo que el sucesor de aplicarle `aNatural` al antecesor de n:

   `aNatural 4` es lo mismo que `Suc (aNatural 3)`

3. *Es hora de definir la función inversa que convierta un natural a un entero. Su firma sería:*

   `aEntero :: Nat ->Int`

   Y aquí hay algunos ejemplos de su ejecución:

```
*Main>aEntero (Suc Cero)
1
*Main>aEntero Cero
0
*Main>aEntero (Suc (Suc (Suc (Suc Cero))))
4
```

   En conjunto las dos funciones anteriores deberían ser equivalentes a la identidad:

```
*Main>aEntero (aNatural 394)
394
*Main>aNatural (aEntero (Suc (Suc (Suc (Suc (Suc Cero))))))
Suc (Suc (Suc (Suc (Suc Cero))))
```

4. *Ahora definamos algunas operaciones para nuestro tipo de dato recursivo Nat. Comenzaremos por la suma cuya firma sería:*

   `sumaNat :: Nat ->Nat ->Nat`

   Y cuya ejecución en terminal se vería del siguiente modo:

```
*Main>sumaNat (Suc (Suc Cero)) (Suc Cero)
Suc (Suc (Suc Cero))
*Main>sumaNat Cero (Suc Cero)
Suc Cero
```

Pista: Para el caso base observa que si sumamos 0 con cualquier otro número x, el resultado será x. Para el caso recursivo, si sumamos el sucesor de un número x con algún otro número z, el resultado es el sucesor de la suma de x con z:

$$\texttt{Suc (6) + 4 = Suc (6 + 4)}$$

Advertencia: No vale definir la función utilizando las funciones de los ejercicios 2 y 3. La idea es realizar una definición recursiva explícita sobre el tipo de dato *Nat*.

Sin embargo podemos utilizar esas funciones para probar el funcionamiento de nuestra suma:

```
*Main>a = aNatural 4
*Main>b = aNatural 9
*Main>a
Suc (Suc (Suc (Suc Cero)))
*Main>b
Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc Cero)))))))
*Main>s = sumaNat a b
*Main>s
Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc Cero)))))))))))
*Main>aEntero s
```

5. *Define una función que multiplique números naturales. ¿Cuál sería su firma de tipo? Puedes utilizar la firma de la suma como inspiración.*

   Veamos su ejecución en un ejemplo:

```
*Main>multNat (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
*Main>a = aNatural 35
*Main>b = aNatural 11
*Main>m = multNat a b
*Main>aEntero m
385
```

   Pista: Pensemos en la relación que existe entre la multiplicación, la suma y el sucesor:

$$5 \times 3 = \mathit{sucesor}(4) \times 3 = 3 + 4 \times 3$$

   Advertencia: Del mismo modo que en `sumaNat` no vale definir `multNat` utilizando `aNatural` ni `aEntero`.

6. Es hora de abordar un nuevo tipo de dato. Vamos a definir **árboles binarios** sobre números enteros con el siguiente dato recursivo:

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

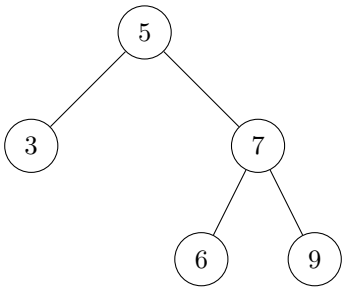
Veamos algunos ejemplos de árboles con el código que los representaría:

Árbol 1:

Código 1:

```
ej1 :: Arbol
ej1 = ( Nodo  Vacio 13  Vacio)
```

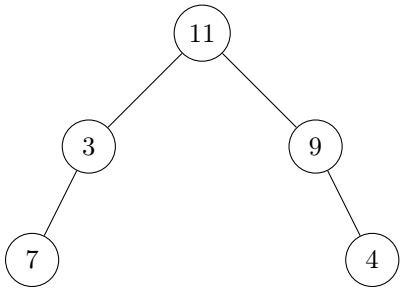
Árbol 2:



Código 2:

```
ej2 :: Arbol
ej2 =  Nodo ( Nodo  Vacio 3  Vacio)
       5
       ( Nodo ( Nodo  Vacio 6  Vacio)
       7
       ( Nodo  Vacio 9  Vacio))
```

Árbol 3:



Código 3:

```
ej3 :: Arbol
ej3 =  Nodo ( Nodo ( Nodo  Vacio 7  Vacio) 3  Vacio)
       11
       ( Nodo  Vacio 9 ( Nodo  Vacio 4  Vacio))
```

Transcribe el tipo de dato Arbol y define tres variables que representen árboles de al menos 4 nodos. Dibuja los árboles que representaste en papel e incluye las imágenes en tu entrega.

7. Por último vamos a definir la función:

   `ocurre :: Int ->Arbol ->Bool`

   que regresa verdadero si el entero dado está en el árbol. Por ejemplo:

```
*Ma<@in>ocurre ej3 9
True
*Main>ocurre ej3 19
False
*Main>multNat (Suc (Suc Cero)) (Suc (Suc (Suc Cero))))
```

Entrega

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

- El archivo `Practica5.hs` con el código indicado en los ejercicios.
- Las imágenes correspondientes a los dibujos de árboles del ejercicio 6.
- Un archivo readMe.text que contenga 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:

Practica05_ Apellido1-Apellido2-Nombre(s).zip

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