DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Prueba de Oposición

Área Teoría

Benjamín Alejandro Scaffidi

Segundo Cuatrimestre 2024

DC - UBA

Primero que nada, el enunciado

Ejercicio 13 ★

Considerar el siguiente tipo, que representa a los árboles binarios:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

I. Usando recursión explícita, definir los esquemas de recursión estructural (`foldAB`) y primitiva (`recAB`), y dar sus tipos.

II. Definir las funciones `esNil`, `altura` y `cantNodos` (para `esNil` puede utilizarse `case` en lugar de `foldAB` o `recAB`).

Donde estamos parados

Los estudiantes ya deberían estar familiarizados con los siguientes conceptos:

- Nociones de Programación Funcional repasadas en la práctica 0
- Haber visto en la teórica:
  - Esquemas de recursión sobre listas
  - Tipos de datos algebraicos
  - Esquemas de recursión sobre tipos de datos algebraicos
- Se supone también que en la clase en la que se dé este ejercicio ya discutieron esquemas de recursión sobre listas

Lo que queremos lograr

- Nos gustaría que los alumnos entiendan el concepto de recursión estructural
- Que no solo entiendan su uso para listas, también sean capaces de reconocerlo y hacer uso del mismo para otros tipos de datos algebraicos
- De esta manera, lograr que entiendan el poder de esta herramienta

Primero que nada, repasito

El enunciado nos pide trabajar con un nuevo tipo ¿Pero cómo era esto?

En Haskell tenemos algunos tipos de datos "básicos", como Char, Int, Float, ...

Además, Haskell nos permite definir nuestros propios tipos de datos mediante la cláusula data de la siguiente manera:

```
data Tipo = ⟨declaración de los constructores⟩
```

Primero que nada, repasito

Un poco mas general, ya vimos que los tipos de datos algebráicos están definidos de la siguiente manera

Tipos de datos algebraico — caso general

En general un tipo de datos algebraico tiene la siguiente forma:

$$
\begin{aligned}
\texttt{data T} = \;& \texttt{CBase}_1\ \langle \textit{parámetros} \rangle \\
& \ldots \\
\mid\; & \texttt{CBase}_n\ \langle \textit{parámetros} \rangle \\
\mid\; & \texttt{CRecursivo}_1\ \langle \textit{parámetros} \rangle \\
& \ldots \\
\mid\; & \texttt{CRecursivo}_m\ \langle \textit{parámetros} \rangle
\end{aligned}
$$

- Los constructores base no reciben parámetros de tipo T.
- Los constructores recursivos reciben al menos un parámetro de tipo T.
- Los valores de tipo T son los que se pueden construir aplicando constructores base y recursivos un número **finito** de veces y **sólo** esos.

Primero que nada, repasito

Y vimos que su esquema de recursión estructural se da por

La recursión estructural se generaliza a tipos algebraicos en general.
Supongamos que `T` es un tipo algebraico.
Dada una función `g :: T -> Y` definida por ecuaciones:

$$
\begin{aligned}
&\texttt{g (CBase}_1\ \langle \textit{parámetros} \rangle\texttt{)} && = && \langle \textit{caso base}_1 \rangle \\
&\ldots \\
&\texttt{g (CBase}_n\ \langle \textit{parámetros} \rangle\texttt{)} && = && \langle \textit{caso base}_n \rangle \\
&\texttt{g (CRecursivo}_1\ \langle \textit{parámetros} \rangle\texttt{)} && = && \langle \textit{caso recursivo}_1 \rangle \\
&\ldots \\
&\texttt{g (CRecursivo}_m\ \langle \textit{parámetros} \rangle\texttt{)} && = && \langle \textit{caso recursivo}_m \rangle
\end{aligned}
$$

Decimos que g está dada por **recursión estructural** si:

1. Cada caso base devuelve un valor fijo.
2. Cada caso recursivo se escribe combinando:
   - Los parámetros del constructor que no son de tipo `T`.
   - El llamado recursivo sobre cada parámetro de tipo `T`.

   Pero:
   - Sin usar los parámetros del constructor que son de tipo `T`.
   - Sin hacer a otros llamados recursivos.

40

Ahora sí, a pensar

Ejercicio 13 ★

Considerar el siguiente tipo, que representa a los árboles binarios:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

I. Usando recursión explícita, definir los esquemas de recursión estructural (`foldAB`) y primitiva (`recAB`), y dar sus tipos.

II. Definir las funciones `esNil`, `altura` y `cantNodos` (para `esNil` puede utilizarse `case` en lugar de `foldAB` o `recAB`).

- Primero que nada pensemos en el foldAB

Ahora sí, a pensar

- Vemos que foldAb tiene dos constructores, uno recursivo y uno base

Ahora sí, a pensar

- Vemos que el tipo tiene dos constructores, uno recursivo y uno base
- La función debería ser capaz de trabajar con cada uno, recibir un AB, y devolver algo
- Entonces el tipo de la función debería ser algo del estilo:
- $foldAB :: ? \to ? \to ? \to ?$
- Pensemos ahora como podemos completarlo

Ahora sí, a pensar

- Por un lado, vamos a querer que nuestra función devuelva un valor en específico, por ahora digamos que es de tipo b, y que modele la recursión estructural sobre un AB
- Entonces el tipo de la función debería ser algo del estilo:
- $foldAB :: ? \to ? \to \ AB\ a \to \ b$
- Consideremos el caso base. ¿Cuántos parámetros tiene?

Ahora sí, a pensar

- Muy bien, ninguno! Vamos a querer que el tipo de la función asociada con este caso sea una constante
- Entonces el tipo de la función debería ser algo del estilo:
- $foldAB :: \ b \to ? \to \ AB \ a \to \ b$
- Ahora pensemos el caso recursivo. ¿Cuántos parámetros tiene?

Ahora sí, a pensar

- Ahora tenemos tres parámetros! (¿Cuántos de ellos son recursivos?)
- Queremos entonces que la función que use nuestro fold sea capaz de tomar el resultado de haber hecho recursión estructural sobre cada uno de los parámatros (¿De qué tipo debería ser el resultado de esas recursiones?), así como también el parámetro no recursivo
- Entonces el tipo de la función debería ser algo del estilo:
- $foldAB :: \ b \to (b \to a \to b \to b) \to AB\ a \to b$
- Tenemos el tipo de nuestra función! ahora tenemos que definirla

Ahora sí, a pensar

- Partiendo del ejemplo anterior, el esqueleto de la definición debería ser
- Como dijimos antes, el caso base debería ser una simple función constante, por lo que deberíamos tener algo como:

$$foldAB :: \ b \to (b \to a \to b \to b) \to AB\ a \to b$$

$$foldAB\ cNil\ cBint\ =\ case\ t\ of$$

$$Nil \to$$

$$Bin\ (i\ r\ d) \to$$

Ahora sí, a pensar

- Como dijimos antes, el caso base debería ser aplicar la funciuna función constante, por lo que

$$foldAB :: \; b \to (b \to a \to b \to b) \to AB\ a \to b$$

$$foldAB\ cNil\ cBint\ =\ case\ t\ of$$

$$Nil \to cNil$$

$$Bin\ (i\ r\ d) \to$$

Ahora sí, a pensar

- Como dijimos antes, el caso base debería ser una función constante, por lo que
- Ahora queda el caso recursivo, deberíamos pasarle a la función el parámetro no recursivo, así como los resultados de la recursión

$$foldAB :: \ b \to (b \to a \to b \to b) \to AB\ a \to b$$

$$foldAB\ cNil\ cBint\ =\ case\ t\ of$$

$$Nil \to cNil$$

$$Bin\ (i\ r\ d) \to cBin\ (foldAB\ cNil\ cBin\ i)\ r\ (foldAB\ cNil\ cBin\ d)$$

Ahora sí, a pensar

- Bien! Parecería que nuestra solución ya está...
- Pero es medio ilegible el caso recursivo
- Podemos usar el where!

$$foldAB :: \ b \to (b \to a \to b \to b) \to AB\ a \to b$$

$$foldAB\ cNil\ cBint\ =\ case\ t\ of$$

$$Nil \to cNil$$

$$Bin\ (i\ r\ d) \to cBin\ (rec\ i)\ r\ (rec\ d)$$

$$where\ rec = foldAB\ cNil\ cBin$$

Ahora sí, a pensar

- Ahora tenemos que pensar en como implementar recursión primitiva para esta estructura
- Recordemos como se veía en listas

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []       = ⟨caso base⟩
g (x : xs) = ⟨caso recursivo⟩
```

Decimos que la definición de g está dada por **recursión primitiva** si:

1. El caso base devuelve un valor fijo `z`.
2. El caso recursivo se escribe usando (cero, una o muchas veces) `x`, `(g xs)` y también `xs`, pero sin hacer otros llamados recursivos.

```
g []       = z
g (x : xs) = ... x ... xs ... (g xs) ...
```

Similar a la recursión estructural, pero permite referirse a `xs`.

Ahora sí, a pensar

- Podemos observar que lo que define a este tipo de recursión es el acceso al parámetro recursivo
- Con recursión estructural sólo conocíamos el resultado de la recursión

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []       = ⟨caso base⟩
g (x : xs) = ⟨caso recursivo⟩
```

Decimos que la definición de g está dada por **recursión primitiva** si:

1. El caso base devuelve un valor fijo `z`.
2. El caso recursivo se escribe usando (cero, una o muchas veces) `x`, `(g xs)` y también `xs`, pero sin hacer otros llamados recursivos.

```
g []       = z
g (x : xs) = ... x ... xs ... (g xs) ...
```

Similar a la recursión estructural, pero permite referirse a `xs`.

Ahora sí, a pensar

- Pensemos ahora como quedaría este tipo de recursión para nuestra estructura
- En un principio vamos a tener que pensar en el tipo de la función
- El caso base, así como la estructura sobre la cual se va a hacer la recursión y el tipo que devolvemos, deberían mantenerse
- Tenemos que considerar el tipo de la función a tomar para el caso recursivo
- Como dijimos antes, ahora también va a recibir los parámetros recursivos

$recAB :: \ b \to ? \to AB \ a \to b$

Ahora sí, a pensar

- Pensemos ahora como quedaría este tipo de recursión para nuestra estructura
- En un principio vamos a tener que pensar en el tipo de la función
- El caso base, así como la estructura sobre la cual se va a hacer la recursión y el tipo que devolvemos, deberían mantenerse
- Tenemos que considerar el tipo de la función a tomar para el caso recursivo
- Como dijimos antes, ahora también va a recibir los parámetros recursivos

$$recAB :: \ b \to (AB\ a \to a \to AB\ a \to b \to b \to b) \to AB\ a \to b$$

Ahora sí, a pensar

- Ya tenemos el tipo, ahora queda pensar en la definición
- El caso base debería ser el mismo, por lo que solo queda el recursivo
- Como venimos discutiendo, vamos a querer que ahora tome los parámetros recursivos

$$recAB :: \; b \to (AB\ a \to a \to AB\ a \to b \to b \to b) \to AB\ a \to b$$

$$recAB\ cNil\ cBint\ =\ case\ t\ of$$

$$Nil \to cNil$$

$$Bin\ (i\ r\ d) \to cBin\ i\ r\ d\ (rec\ i)\ (rec\ d)$$

$$where\ rec = recAB\ cNil\ cBin$$

Ahora sí, a pensar

- Ya tenemos los esquemas de recursión implementados, ahora quedan los otros ejercicios
- Al pizarrón!

Gracias por su tiempo!