Capítulo 3
Álgebra de Tablas

Trabajo con Listas

¿Por qué trabajar con listas?

- Una forma de ver una **tabla** es como una **lista de tuplas** (con el mismo esquema).
- Veremos que dadas varias tablas de la base de datos, una **consulta** se puede expresar como **composición de operaciones sobre tablas**.
- Por lo tanto para hacer consultas necesitamos poder definir **operaciones sobre listas**.

¿Por qué trabajar con listas?

- **Ventajas de trabajar con listas:**
  - **Permiten modelar tablas como en SQL** en el sentido que podemos tener tuplas repetidas, podemos ordenar las tuplas.
  - El **álgebra relacional** no permite esto porque usa **tablas** que son **conjuntos de tuplas** (entonces no se permiten tuplas repetidas, ni se pueden consultar los resultados de manera ordenada, etc.)

Un poco de notación

- Usaremos para definir listas y operaciones sobre listas un poco del conocimiento y la notación de programación funcional.
- **Un poco de notación:**
  - La lista vacía se denota con: []
  - usamos el operador : que agrega un elemento a una lista.
  - Por ejemplo: [a, b, c] = a : b: c : []
  - [T] se usa para indicar el conjunto de las listas de tipo T.
  - Por ejemplo: [int], [string], etc.

Recursiones sobre listas

- Muchas funciones sobre listas se pueden definir usando **recursión**.
- **Ejemplo**:

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

- Usando operaciones sobre listas definidas y operaciones básicas, podemos **evaluar** una expresión sobre listas.

Funciones sobre listas

- Ejemplo: Evaluar suma[1,2,3]

Suma 1 : 2 : 3 : []

= {suma.3} 1 + suma 2 : 3 : []

= **¿qué va aquí?**

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

Funciones sobre listas

- Ejemplo: Evaluar suma[1,2,3]

Suma 1 : 2 : 3 : []

= {suma.3} 1 + suma 2 : 3 : []

= {suma.3} 1 + 2 + suma 3:[]

= {suma.3} 1 + 2 + 3 + suma []

= **¿Cómo acaba?**

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

Funciones sobre listas

- Ejemplo: Evaluar suma[1,2,3]

Suma 1 : 2 : 3 : []

= {suma.3} 1 + suma 2 : 3 : []

= {suma.3} 1 + 2 + suma 3:[]

= {suma.3} 1 + 2 + 3 + suma []

= {suma.2} 1 + 2 + 3 + 0

= {+} 6

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

Recursiones sobre listas

- La operación Sum que hicimos es un ejemplo de **definición por recursión estructural sobre listas**.
- Gran parte de las operaciones sobre listas que veremos son recursiones de este tipo.
- Estas definiciones tienen la forma:

  f [] = c

  f x : xs = h x (f xs)

- Donde:

  f:: [a] -> b

  c:: b

  h:: a -> b -> b

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

¿Qué sería c y h aquí?

Recursiones sobre listas

- La operación Sum que hicimos es un ejemplo de **definición por recursión estructural sobre listas**.
- Gran parte de las operaciones sobre listas que veremos son recursiones de este tipo.
- Estas definiciones tienen la forma:

  f [] = c

  f x : xs = h x (f xs)

- Donde:

  f:: [a] -> b

  c:: b

  h:: a -> b -> b

1. Suma :: [Int] -> Int
2. Suma [] = 0
3. Suma x: xs = x + suma xs

aquí
c = 0
h = +

Funciones de alto orden

- Una **función de alto orden** porque se define sobre funciones además de sobre listas.
  - Por ejemplo: map, foldr, etc.
- Una función de alto orden es **map**:
  - map :: (a -> b) -> list a -> list b
  - **¿Cómo definir map recursivamente?**

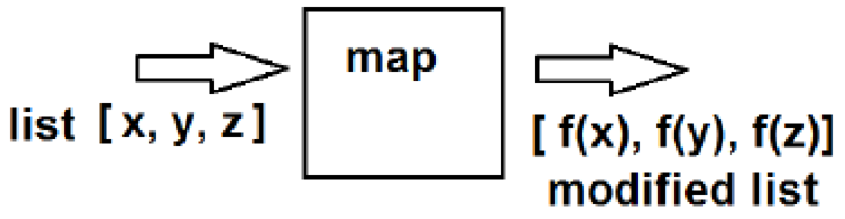

Figure 2: Despliegue de la función map.

Funciones de alto orden

- Una **función de alto orden** porque se define sobre funciones además de sobre listas.
  - Por ejemplo: map, foldr, etc.
- Una función de alto orden es **map**:
  - map :: (a -> b) -> list a -> list b
  - map f [] = []
  - map f x: xs = (f x) : map(xs)

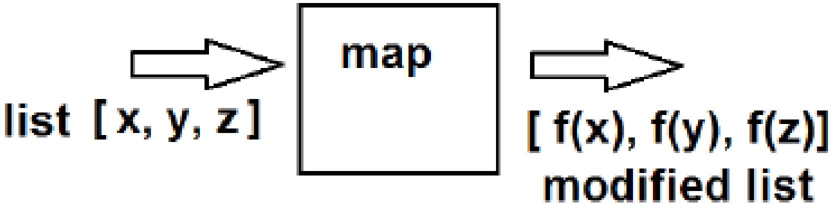

Figure 2: Despliegue de la función map.

- Las funciones de alto orden son potentes en el sentido que permiten dar definiciones de funciones compactas.

Funciones sobre listas

- **Ahora vemos algunas operaciones que vamos a usar en el álgebra de tablas.**
- **Ejercicio**: Describir la función que chequea si un elemento pertenece a una lista.

  -∈- :: a -> [a] -> Bool

Funciones sobre listas

- **Ahora vemos algunas operaciones que vamos a usar en el álgebra de tablas.**
- **Ejercicio**: Describir la función que chequea si un elemento pertenece a una lista.

  -∈- :: a -> [a] -> Bool

  1. x ∈[] = False
  2. $x \in y : xs = x == y \,||\, x \ \in xs$

- **Ejercicio**: Definir la **concatenación** de listas.

  ++ : [a] -> [a] -> [a]

Funciones sobre listas

- **Ahora vemos algunas operaciones que vamos a usar en el álgebra de tablas.**
- **Ejercicio**: Describir la función que chequea si un elemento pertenece a una lista.

  -∈- :: a -> [a] -> Bool

  1. x ∈[] = False
  2. $x \in y : xs = x == y \,||\, x \; \in xs$

- **Ejercicio**: Definir la **concatenación** de listas.

  ++ : [a] -> [a] -> [a]

  [] ++ l' = l'

  (x : l) ++ l' = x : (l ++ l')

Principio de inducción sobre listas

- Probar que: l ++ [] = l

l = []:   [] ++ [] = []

1. l = x:xs
2. (x:xs) ++ [] = x : (xs ++ []) =? x:xs

++ : [a] -> [a] -> [a]

[] ++ l' = l'

(x : l) ++ l' = x : (l ++ l')

- Esta forma de pensar es válida usando lo que se llama inducción.

Principio de inducción sobre listas

- **Definición** (**Principio de inducción sobre listas**). Sea *P* una propiedad sobre listas (notaremos P(l) para indicar que *P* se cumple para la lista l). El principio de inducción sobre listas se define como:

$$\forall l :: [a], P(l) \triangleq P([]) \wedge (\forall x :: a, \forall l' :: [a], P(l') \Longrightarrow P(x : l')$$

++ : [a] -> [a] -> [a]

[] ++ l' = l'

(x : l) ++ l' = x : (l ++ l')

P(l) : l ++ [] = l

P[]: {++ def 1} [] ++ [] = []

HI xs ++ [] = xs

Paso inductive: (x:xs) ++ [] = {++ def 2} x : (xs ++ []) = {por HI} x:xs

Principio de inducción sobre listas

- **Ejercicio**: Probar que ++ es asociativa.
- **¿Cuál es la propiedad a probar?**

Principio de inducción sobre listas

- **Ejercicio**: Probar que ++ es asociativa.
- P(l) : (l ++ l’) ++ l’’ = l ++ (l’ ++ l’’)
- **Caso base**: ([] ++ l’) ++ l’’ = … **¿Cómo sigue?**

[] ++ l’ = l’

(x : l) ++ l’ = x : (l ++ l’)

Principio de inducción sobre listas

- **Ejercicio**: Probar que ++ es asociativa.
- P(l) : (l ++ l') ++ l'' = l ++ (l' ++ l'')
- **Caso base**: ([] ++ l') ++ l'' = {++ def 1} l' ++ l'' = {++ def 1} = [] ++ (l' ++ l'')
- **HI**: (xs ++ l') ++ l'' = xs ++ (l' ++ l'')
- **Paso inductivo.**

  ((x:xs) ++ l') ++ l'' = **¿cómo sigue?**

[] ++ l' = l'

(x : l) ++ l' = x : (l ++ l')

Principio de inducción sobre listas

- **Ejercicio**: Probar que ++ es asociativa.
- P(l) : (l ++ l’) ++ l’’ = l ++ (l’ ++ l’’)
- **Caso base**: ([] ++ l’) ++ l’’ = {++ def 1} l’ ++ l’’ = {++ def 1} = [] ++ (l’ ++ l’’)
- **HI**: (xs ++ l’) ++ l’’ = xs ++ (l’ ++ l’’)
- **Paso inductivo.**

((x:xs) ++ l’) ++ l’’ = {++ def 2} (x : (xs ++ l’)) ++ l’’

= {++ def 2} x : ((xs ++ l’) ++ l’’)

= **¿Cómo sigue?**

[] ++ l’ = l’

(x : l) ++ l’ = x : (l ++ l’)

Principio de inducción sobre listas

- **Ejercicio**: Probar que ++ es asociativa.
- P(l) : (l ++ l’) ++ l’’ = l ++ (l’ ++ l’’)
- **Caso base**: ([] ++ l’) ++ l’’ = {++ def 1} l’ ++ l’’ = {++ def 1} = [] ++ (l’ ++ l’’)
- **HI**: (xs ++ l’) ++ l’’ = xs ++ (l’ ++ l’’)
- **Paso inductivo.**

((x:xs) ++ l’) ++ l’’ = {++ def 2} (x : (xs ++ l’)) ++ l’’

= {++ def 2} x : ((xs ++ l’) ++ l’’)

= {por HI} x: (xs ++ (l’ ++ l’’))

= {++ def 2} (x: xs) ++ (l’ ++ l’’)

[] ++ l’ = l’

(x : l) ++ l’ = x : (l ++ l’)

Funciones sobre listas

- **Ejercicio**: Sea **reverse** sobre listas:

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]
```

- Probar por inducción: reverse (xs ++ ys) = reverse ys ++ reverse xs
- **Queda como tarea.**