Mecanismo de recursión T6

Recursión

Una función recursiva es aquella que en su definición se invoca a sí misma. La misma por lo general cuenta con una definición recursiva y *al menos un* caso base que corta la recursividad.

La recursión es muy importante en Haskell ya que, al contrario que en los lenguajes imperativos, realizamos cálculos declarando *como es* algo, en lugar de declarar como obtener algo. Por este motivo no hay bucles while o bucles for en Haskell y en su lugar tenemos que usar la recursión para declarar como es algo.

Ej:

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0 = n * factorial (n-1)
```

¿Podemos definirla usando `otherwise`?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión `factorial 3`?

```
factorial 3 ⇝ 3 * factorial 2 ⇝ 3 * 2 * factorial 1 ⇝
⇝ 6 * factorial 1 ⇝ 6 * 1 * factorial 0 ⇝ 6 * factorial 0 ⇝
⇝ 6 * 1 ⇝ 6
```

Propiedades de una definición recursiva

- Las llamadas recursivas tienen que "acercarse" a un caso base.
- Tiene que tener uno o más casos base que dependerán del tipo de llamado recursivo. Un caso base es una expresión que no tiene paso recursivo.

Pensar recursivamente

- Casos bases: identificar el o los casos bases
- Casos recursivos: suponiendo que la llamada recursiva es correcta, ¿qué tengo que hacer para completar la solución?

Cuando queremos resolver un problema de forma recursiva, primero pensamos *donde no* se aplica una solución recursiva y si podemos utilizar esto como un *caso base*. Luego pensamos en las *identidades*, por donde deberíamos romper los parámetros (por ejemplo, las lista se rompen en cabeza y cola) y en que parte deberíamos aplicar la función recursiva.

Inducción vs Recursión

- Probar por inducción $P(n) : \sum_{i=1}^{n}(2i-1) = n^2$
- Vale para $n = 1 : \sum_{i=1}^{1}(2i-1) = 1^2$
- Supongo que vale $P(n)$, quiero probar $P(n+1)$
- ¿Qué relación hay entre $\sum_{i=1}^{n}(2i-1)$ y $\sum_{i=1}^{n+1}(2i-1)$?

$$\sum_{i=1}^{n+1}(2i-1) = \left(\sum_{i=1}^{n}(2i-1)\right) + 2n+1$$

- Uso la Hipótesis Inductiva $P(n)$:

$$\sum_{i=1}^{n+1}(2i-1) = n^2 + 2n + 1 = (n+1)^2$$

- ¡¿Pero cómo?! ¡¿Estoy usando lo que quiero probar?!
- Ah, claro... vale $P(1)$ y $P(n) => P(n+1)$, entonces ¡vale para todo n!

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base en Haskell: `f 1 = 1`
- Supongo que ya sé calcular $f(n-1)$, quiero calcular $f(n)$
- ¿Qué relación hay entre $\sum_{i=1}^{n-1}(2i-1)$ y $\sum_{i=1}^{n}(2i-1)$?

$$\sum_{i=1}^{n}(2i-1) = \left(\sum_{i=1}^{n-1}(2i-1)\right) + 2n - 1$$

- Uso la función que sé calcular: $f(n) = f(n-1) + 2n - 1$

  En Haskell: `f n = f (n-1) + 2*n - 1`

- ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?!
- Ah, claro... está definido $f(1)$ y con $f(n-1)$ sé obtener $f(n)$, entonces ¡puedo calcular f para todo n!

Generalización de funciones

En ocasiones, requeriremos solucionar un problema más grande (generalizar) para poder solucionar un problema más chico.

Ej:
Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

$$\begin{array}{l} \text{problema } \mathit{sumaDivisores}(n : \mathbb{Z}) : \mathbb{Z} \ \{ \\ \quad \text{requiere: } \{n > 0\} \\ \quad \text{asegura: } \{res = \sum_{i=1}^{n} \text{ if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\} \\ \} \end{array}$$

¿alcanza con hacer recursión sobre n?
No hay ninguna relaci´on sencilla entre sumaDivisores n y sumaDivisores (n-k) (para ningún k particular)

Quiero sumar los números de 1 hasta n que dividen a n y el resto es 0.
Ej n=4

- [x] 4 mod 1 = 0 => sumo 4 + 2 + 1
- [x] 4 mod 2 = 0 => sumo 4 + 2
- [ ] 4 mod 3 = 1 => sumo 4 + 0
- [x] 4 mod 4 = 0 => sumo 0 +4

¿Qué sucede si definimos primero una funcion más general que devuelve la suma de los divisores de un número hasta cierto punto?

$$\begin{array}{l} \text{problema } \mathit{sumaDivisoresHasta}(n : \mathbb{Z}, k : \mathbb{Z}) : \mathbb{Z} \ \{ \\ \quad \text{requiere: } \{(n > 0) \wedge (k > 0)\} \\ \quad \text{asegura: } \{res = \sum_{i=1}^{k} \text{ if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\} \\ \} \end{array}$$

Podemos definir esta función en haskell recursivamente:

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
sumaDivisoresHasta n 1 = 1
sumaDivisoresHasta n i | (mod n i == 0) = i + sumaDivisoresHasta n (i-1)
                       | otherwise = sumaDivisoresHasta n (i-1)
```

Ahora podemos definir la función sumaDivisores

```
sumaDivisores :: Integer -> Integer
sumaDivisores n = sumaDivisoresHasta n n
```

Recursión en más de un parámetro

Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i^j$$

Veamos primero la especificación:

problema *sumatoriaDoble*$(n : \mathbb{Z}, m : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{(n > 0) \land (m > 0)\}$
  asegura: $\{res = \sum_{i=1}^{n} \sum_{j=1}^{m} i^j\}$
}

Pregunta clave: ¿alcanza con hacer recursión sobre n?

¿Qué sucede si definimos primero una funcion **más específica** que devuelve la sumatoria interna?

```
sumatoriaInterna :: Integer -> Integer -> Integer
```

Ahora parece más sencillo definir `sumatoriaDoble n m` utilizando `sumatoriaInterna n m`. ¿Cómo lo hacemos?

Recursión en más de un parámetro

Veamos cómo sería la especificación:

problema *sumatoriaInterna*$(n : \mathbb{Z}, m : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{(n > 0) \land (m > 0)\}$
  asegura: $\{res = \sum_{j=1}^{m} n^j\}$
}

Ahora podemos definir esta función en Haskell recursivamente

```
sumatoriaInterna :: Integer -> Integer -> Integer
sumatoriaInterna _ 0 = 0
sumatoriaInterna n j = n^j + sumatoriaInterna n (j-1)
```

¿Y por último, cómo definimos `sumatoriaDoble` utilizando lo anterior?

```
sumatoriaDoble :: Integer -> Integer -> Integer
sumatoriaDoble 0 _ = 0
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatoriaInterna n m
```

Entonces, `sumatoriaDoble`, ¿cuántas recursiones involucra?