Capítulo 3

Recursión

La recursión es un *estilo autoreferencial* de definición usado tanto en matemática como en informática. Es una herramienta de programación fundamental, particularmente importante para manipular estructuras de datos. La idea en recursión es dividir un problema en dos casos: si el problema es "fácil" se especifica una solución directa, pero si es "difícil" procedemos a través de varios pasos: primero se redefine el problema desde el punto de vista de un problema más fácil dejando de lado el problema actual temporalmente, así resolvemos el problema más fácil y finalmente utilizamos la solución del problema más fácil para calcular el resultado final. La estrategia de dividir un problema difícil en problemas más fáciles se llama *divide y vencerás* , y es muy útil en el diseño de algoritmos.

La función factorial proporciona un buen ejemplo del proceso descrito anteriormente. La función factorial puede definirse mediante el uso de la notación clara pero informal '. . .' de la siguiente manera:

$$n! = 1 \times 2 \times \cdots \times n$$

Esta definición es correcta, pero depende de la comprensión del lector de la notación '· · · ' para entender lo que ésta significa. Una definición informal como ésta no es la adecuada para proporcionar teoremas y no es un programa ejecutable en la mayoría de los lenguages de programación[1]. Una definición recursiva más precisa de factorial consiste en el siguiente par de ecuaciones:

$$0! = 1$$
$$(n+1)! = (n + 1) \times n !$$

En Haskell, esto se escribiría de la siguiente manera:

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

La primera ecuación especifica el caso fácil, en el cual el argumento es 0 y el resultado es 1, que puede ser devuelto directamente. La segunda ecuación especifica el caso difícil, en el cual el argumento es $n + 1$. Primero la función selecciona un problema ligeramente más fácil para resolver, que es del tamaño n; luego soluciona $n!$ a través de la evaluación de `factorial n`, y multiplica el valor de $n!$ por $n + 1$ para obtener el resultado final.

Las definiciones recursivas consisten en una colección de ecuaciones que establecen propiedades de la función que se define. Por ejemplo, existen numerosas propiedades algebraicas de la función factorial, y una de ellas es utilizada como la segunda ecuación de la definición recursiva. Los lenguajes de programación que permiten este estilo de definición son llamados generalmente *lenguajes de programación declarativa*, porque un programa consiste en un conjunto de declaraciones de propiedades. El programador debe encontrar un conjunto adecuado de propiedades para declarar, generalmente vía ecuaciones recursivas y su implementación en un lenguaje de programación, y luego encontrar la forma de resolver las ecuaciones. Lo opuesto al lenguaje de programación declarativa es un *lenguaje de programación imperativa*, en el cual se dan una serie de órdenes que, al ser obedecidas, resultarán en el cálculo del resultado.

---

1 Haskell es un lenguaje de programación de muy alto nivel y permite este estilo: en Haskell se puede definir la función `factorial n = product [1..n]`. Sin embargo, la mayoría de los lenguajes de programación no permiten esto, e incluso Haskell trata la notación `[1..n]` como una abreviación de alto nivel que se ejecuta internamente usando recursión.

3.1 Recursión sobre listas

Para las funciones recursivas sobre listas, el caso "fácil" es la lista vacía `[]` y el caso "difícil" es una lista no vacía, que puede ser escrita `x:xs`. Una función recursiva sobre listas tiene la siguiente forma:

`f :: [a] –>` *tipo de resultado*
`f []` = *resultado de lista vacía*
`f (x:xs)` = *resultado definido usando (`f xs`) y `x`*

Un ejemplo simple es la función `length` , que cuenta los elementos en una lista; por ejemplo, `length [1,2,3]` es `3` y una forma de definir la función es:

```
length :: [a]   –>  Int
length []        = 0
length (x:xs) = 1 + length xs
```

Una lista vacía no contiene elementos, por lo tanto `length []` devuelve 0. Cuando la lista contiene al menos un elemento, la función calcula la longitud del resto de la lista evaluando `length xs` y luego suma 1 para obtener la longitud de la lista completa. Podemos desarrollar la evalución de `length [1,2,3]` usando el razonamiento ecuacional. Este proceso es similar al de resolver un problema de álgebra, y el programa Haskell realiza esencialmente el mismo cálculo cuando se ejecuta. Simplificar una ecuación a través del razonamiento ecuacional es la forma correcta de "ejecutar paso a paso" un programa funcional. Cuando trabajamos con un ejemplo asi, debemos recordar que `[1,2,3]` es una notación reducida para `1:(2:(3:[]))`.

```
length [1,2,3]
= 1 + length [2,3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 3
```

Es mejor pensar en la recursión como un cálculo sistemático que tratar de imaginar operaciones o subrutinas de bajo nivel en la computadora. Los libros de texto sobre lenguajes de programación imperativa algunas veces explican la recursión a través de la implementación del lenguaje subyacente. En un lenguaje funcional, la recusión debe ser evaluada desde un alto nivel, como una técnica ecuacional, y los niveles de un bajo nivel deben ser dejados para el compilador.

La función `sum` ofrece un ejemplo similar de recursión. Esta función suma los elementos de una lista; por ejemplo, `sum [1,2,3]` devuelve 1+2+3 = 6 y puede ser definida de la siguiente manera:

```
sum :: Num a => [a] –> a
sum [] = 0
sum (x:xs) = x + sum xs
```

Esta definición refleja el hecho de que los elementos de la lista que toma como argumento deben ser números. El tipo de contexto '`Num a =>`' dice que `a` puede representar cualquier tipo para el cual las operaciones numéricas estén definidas. Por lo tanto `sum` puede ser aplicada a una lista de enteros de 32 bits o a una lista de números con punto flotante, entre otros.

La definición tiene la misma forma que la del ejemplo anterior. Definimos la suma de una lista vacía como 0; esto se necesita para que la recursión funcione apropiadamente y que termine en el caso base. Esto también le otorga buenas propiedades algebraicas a la función, por ejemplo, esta es la razón por la cual 0 ! se define 1. Para el patrón recursivo, sumamos el primer elemento de la lista `x` a la suma del resto de los elementos, calculada a través de la llamada recursiva `sum xs`. El valor de `sum [1,2,3]` puede ser calculado usando razonamiento ecuacional:

```
sum [1,2,3]
= 1 + sum [2,3]
```

```
= 1 + (2 + sum [3])
= 1 + (2 + (3 + sum []))
= 1 + (2 + (3 + 0))
= 6
```

Hasta ahora, las funciones que hemos definido reciben una lista y devuelven un número. Ahora consideremos funciones que toman listas y devuelvan listas. Un ejemplo típico es la función de concatención de listas (cuyo operador en Haskell es ++). Esta función toma dos listas y las junta (concatena) en una lista más grande. Por ejemplo, `[1,2,3] ++ [9,8,7,6]` = `[1,2,3,9,8,7,6]`.

Esta función toma dos listas como argumentos y hace recursión sobre el primer argumento. Es fácil obtener el valor de `[] ++ [a,b,c]`: la primera lista no aporta nada, entonces el resultado es simplemente `[a,b,c]`. Esta observación ofrece un caso base que es esencial para que la recursión funcione: podemos definir `[] ++ ys = ys` y para el caso recursivo debemos considerar una expresión de la forma `(x:xs) ++ ys`. El primer elemento del resultado debe ser el valor `x`. Luego tenemos una lista que consiste de todos los elementos de `xs`, seguidos por todos los elementos de `ys`; esa lista es simplemente `xs++ys`. Esto sugiere la siguiente definición:

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

Trabajar con un ejemplo es una buena forma de asegurar la comprensión de la definición:

```
[1,2,3] ++ [9,8,7,6]
= 1 : ([2,3] ++ [9,8,7,6])
= 1 : (2 : ([3] ++ [9,8,7,6]))
= 1 : (2 : (3 : ([] ++ [9,8,7,6])))
= 1 : (2 : (3 : [9,8,7,6]))
= 1 : (2 : [3,9,8,7,6])
= 1 : [2,3,9,8,7,6]
= [1,2,3,9,8,7,6]
```

Una vez que sabemos la estructura de la definición (recursión sobre `xs`) es directo crear las ecuaciones. Lo más difícil de escribir la definición de `++` es decidir sobre qué lista hacer la recursión. Obsérvese que la definición solo trata `ys` como un valor ordinario, y nunca comprueba si está vacía o no. Pero no siempre se puede asumir que, si existen varias listas de argumento, la recursión se hará sobre la primera. Algunas funciones hacen recursión sobre el segundo argumento y no sobre el primero, y algunas funciones hacen recursión simultáneamente sobre *varias* listas.

La función `zip` es un ejemplo de una función que toma dos listas y hace recursión simultáneamente sobre ambas. Esta función toma dos listas, y devuelve una lista de pares de elementos (tuplas). En cada par, el primer valor proviene de la primera lista y el segundo valor de la segunda lista. Por ejemplo, `zip [1,2,3,4] ['A', '*', 'q', 'x']` devuelve `[(1,'A'), (2,'*'), (3,'q'), (4,'x')]`. Notar que las dos listas pueden tener distintas longitudes; en ese caso, el resultado tendrá la misma longitud que la del argumento más corto. Por ejemplo, `zip [1,2,3,4] ['A', '*', 'q']` devuelve sólo `[(1,'A'), (2,'*'), (3,'q')]` porque no hay nada para formar pareja con el elemento 4.

La definición de `zip` debe hacer recursión sobre *ambas* listas de argumentos, porque las dos listas tienen que mantenerse en sincronía. Por lo tanto, existen dos casos bases, porque es posible que cualquiera de las listas sea vacía pero no hay necesidad de escribir un tercer caso base donde anmbas sean vacías a la vez, es decir el caso `zip [] [] = []` no es necesario. Luego la definición es:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Aquí tenemos el cálculo del ejemplo citado anteriormente. La recursión termina cuando la segunda lista esté

vacía; la ecuación del segundo caso base define el resultado de `zip [4] []`, que será `[]` a pesar de que el primer argumento no esté vacío.

```
zip [1,2,3,4] ['A', '*', 'q']
= (1,'A') : zip [2,3,4] ['*', 'q']
= (1,'A') : ((2,'*') : zip [3,4] ['q'])
= (1,'A') : ((2,'*') : ((3,'q') : zip [4] []))
= (1,'A') : ((2,'*') : ((3,'q') : []))
= (1,'A') : ((2,'*') : [(3,'q')])
= (1,'A') : [(2,'*'), (3,'q')]
= [(1,'A'), (2,'*'), (3,'q')]
```

Otro ejemplo, es la función `concat` toma una lista de listas (`[[a]]`) y la convierte en una lista de elementos (`[a]`), es decir, que "aplana" la estructura de la lista original. Por ejemplo, `concat [[1], [2,3], [4,5,6]]` devuelve `[1,2,3,4,5,6]`.

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

Al trabajar con los cálculos de ejemplo, directamente simplificamos todas las aplicaciones de `(++)`. Por supuesto que cada una de esas aplicaciones lleva consigo otra recursión, que es similar a los ejemplos de `(++)` presentados anteriormente.

```
concat [[1], [2,3], [4,5,6]]
= [1] ++ concat [[2,3], [4,5,6]]
= [1] ++ ([2,3] ++ concat [[4,5,6]])
= [1] ++ ([2,3] ++ [4,5,6])
= [1] ++ [2,3,4,5,6]
= [1,2,3,4,5,6]
```

El caso base para la función que construye una lista debe devolver una lista, y esto, por lo general, es simplemente `[]`. El caso recursivo construye una lista al adjuntar un valor al resultado que devuelve el llamado recursivo.

Al definir una función recursiva `f`, es importante que la recursión trabaje en una lista más corta que el argumento original de `f`. Por ejemplo, en la aplicación `sum [1,2,3]`, la recursión calcula `sum [2,3]`, cuyo argumento es un elemento más corto que el argumento original `[1,2,3]`. Si una función fuese definida en forma incorrecta, con una recursión más grande que el problema original, entonces podría entrar en un lazo infinito y no terminar ni devolver resultado alguno.

Todos los ejemplos que hemos visto hasta aquí realizan la recursión en una lista más corta que el argumento original, generalmente tomamos `x:xs` como argumento original y realizamos la recusión sobre `xs`, la cual contiene un elemento menos que la lista `x:xs`. Sin embargo, dado que la recursión está resolviendo un problema menor que el original, el caso recursivo puede ser cualquiera, no necesariamente tiene que trabajar en la cola de la lista original. A menudo un buen enfoque es tratar de dividir el problema en dos, en vez de reducir su tamaño en 1.

Ejercicio 1. Escriba una función recursiva `copia :: [a] -> [a]` que copie su lista de argumentos. Por ejemplo, `copia [2] = [2]`.

Ejercicio 2. Escriba una función `inversa` que tome una lista de pares y cambie los pares de elementos. Por ejemplo, `inversa [(1,2),(3,4)] = [(2,1),(4,3)]`

Ejercicio 3. Escriba una función `fusión :: Ord a => [a] -> [a] -> [a]` que toma dos listas ordenadas y devuelve una lista ordenada que contiene los elementos de cada una.

Ejercicio 4. Escriba (`!!`), una función que tome un número neutro `n` y una lista y selecione el elemento en el

lugar `n` de la lista. Ya que el tipo de `n` no impide ingresar un valor que caiga fuera del rango de índices posibles de la lista, el resultado debería de ser de tipo `Maybe` . Por ejemplo,

```
[1,2,3]!!0 = Just 1
[1,2,3]!!2 = Just 3
[1,2,3]!!5 = Nothing
```