Programación Funcional

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Funciones recursivas

Recursión

- Las variables en un lenguaje funcional son **inmutables**.
- Esto se debe a la no existencia de una sentencia de asignación que permita modificarlas.

Recursión

- Las variables en un lenguaje funcional son **inmutables**.
- Esto se debe a la no existencia de una sentencia de asignación que permita modificarlas.
- Una consecuencia directa de esto es la ausencia de la **iteración** como estructura de control.

Recursión

- Las variables en un lenguaje funcional son **inmutables**.
- Esto se debe a la no existencia de una sentencia de asignación que permita modificarlas.
- Una consecuencia directa de esto es la ausencia de la **iteración** como estructura de control.
- Es entonces que aparece la **recursión** como el mecanismo natural para definir las funciones en programación funcional.

Recursión

- Las variables en un lenguaje funcional son **inmutables**.
- Esto se debe a la no existencia de una sentencia de asignación que permita modificarlas.
- Una consecuencia directa de esto es la ausencia de la **iteración** como estructura de control.
- Es entonces que aparece la **recursión** como el mecanismo natural para definir las funciones en programación funcional.
- En esta clase veremos definiciones recursivas sobre enteros y listas. Más adelante veremos definiciones recursivas para tipos de datos algebraicos en general.

Recursión sobre enteros

Factorial de un número

- La función factorial es un clásico ejemplo de función recursiva.
- Su formulación matemática sobre naturales es bien conocida:

$$0! = 1$$
$$n! = n \; x \; (n - 1)\,!$$

- Tenemos varias alternativas para escribir esta función en Haskell.
- Una de ellas es una formulación no recursiva del estilo de las que vimos en la clase pasada.

$$factorial \; n = product \; [1 \,..\, n]$$

donde *product* realiza el producto de los elementos de la lista.

Factorial de un número

- Definición a través de una ecuación condicional:

$$
\begin{aligned}
&\mathit{fact} :: \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{fact}\ n \mid n == 0 = 1\\
&\qquad\ \ \mid n > 0\ \ = n * \mathit{fact}\ (n - 1)
\end{aligned}
$$

Factorial de un número

- Definición a través de una ecuación condicional:

$$
\begin{aligned}
&\mathit{fact} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{fact}\ n \mid n == 0 = 1 \\
&\qquad\quad \mid n > 0 \;\; = n * \mathit{fact}\ (n-1)
\end{aligned}
$$

- Otra definición:

$$
\begin{aligned}
&\mathit{fact}\ n \mid n == 0 \qquad\;\; = 1 \\
&\qquad\quad \mid \mathit{otherwise} = n * \mathit{fact}\ (n-1)
\end{aligned}
$$

Factorial de un número

- Definición a través de una ecuación condicional:

$$
\begin{aligned}
&\mathit{fact} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{fact}\ n \mid n == 0 = 1 \\
&\qquad\ \ \mid n > 0 \ \ = n * \mathit{fact}\ (n - 1)
\end{aligned}
$$

- Otra definición:

$$
\begin{aligned}
&\mathit{fact}\ n \mid n == 0 \qquad\ = 1 \\
&\qquad\ \ \mid \mathit{otherwise} = n * \mathit{fact}\ (n - 1)
\end{aligned}
$$

- Las definiciones anteriores **no** son equivalentes. Por qué?

Factorial de un número

- Por pattern-matching:

$$\begin{aligned} &\mathit{fact}\ 0 = 1 \\ &\mathit{fact}\ n = n * \mathit{fact}\ (n-1) \end{aligned}$$

Factorial de un número

- Por pattern-matching:

$$\begin{aligned} &fact\ 0 = 1 \\ &fact\ n = n * fact\ (n-1) \end{aligned}$$

- Mediante una definición tail-recursiva que usa un acumulador:

$$\begin{aligned} &fact\ n = factacc\ 1\ n \\ &\quad \textbf{where} \\ &\qquad factacc\ a\ 0 = a \\ &\qquad factacc\ a\ n = factacc\ (a * n)\ (n-1) \end{aligned}$$

Factorial de un número

- Por pattern-matching:

$$fact\ 0 = 1$$
$$fact\ n = n * fact\ (n-1)$$

- Mediante una definición tail-recursiva que usa un acumulador:

$$fact\ n = factacc\ 1\ n$$
where
$$factacc\ a\ 0 = a$$
$$factacc\ a\ n = factacc\ (a*n)\ (n-1)$$

- En todas estas definiciones de factorial estamos asumiendo como pre-condición que $n \geqslant 0$.

Funciones definidas por recursión estructural

- Potencia de 2:

$$\mathit{pot2} :: \mathit{Integer} \rightarrow \mathit{Integer}$$
$$\mathit{pot2}\ 0 = 1$$
$$\mathit{pot2}\ n = 2 * \mathit{pot2}\ (n - 1)$$

Funciones definidas por recursión estructural

- Potencia de 2:

$$
\begin{aligned}
&\mathit{pot2} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{pot2}\ 0 = 1\\
&\mathit{pot2}\ n = 2 * \mathit{pot2}\ (n - 1)
\end{aligned}
$$

- Sumatoria de valores de una función:

$$
\begin{aligned}
&\mathit{sumF} :: (\mathit{Integer} \to \mathit{Integer}) \to \mathit{Integer} \to \mathit{Integer}\\
&\mathit{sumF}\ f\ 0 = f\ 0\\
&\mathit{sumF}\ f\ n = f\ n + \mathit{sumF}\ f\ (n - 1)
\end{aligned}
$$

Funciones definidas por recursión estructural

- Potencia de 2:

$$
\begin{aligned}
&\mathit{pot2} :: \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{pot2}\ 0 = 1\\
&\mathit{pot2}\ n = 2 * \mathit{pot2}\ (n-1)
\end{aligned}
$$

- Sumatoria de valores de una función:

$$
\begin{aligned}
&\mathit{sumF} :: (\mathit{Integer} \rightarrow \mathit{Integer}) \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{sumF}\ f\ 0 = f\ 0\\
&\mathit{sumF}\ f\ n = f\ n + \mathit{sumF}\ f\ (n-1)
\end{aligned}
$$

- Multiplicación

$$
\begin{aligned}
&\mathit{mult} :: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{mult}\ m\ 0 = 0\\
&\mathit{mult}\ m\ n = m + \mathit{mult}\ m\ (n-1)
\end{aligned}
$$

Recursión general

- Números de fibonacci

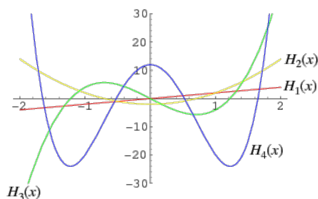
$$\mathit{fib} :: \mathit{Integer} \rightarrow \mathit{Integer}$$
$$\mathit{fib}\ 0 = 0$$
$$\mathit{fib}\ 1 = 1$$
$$\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$$

Recursión general

- Números de fibonacci

$$
\begin{aligned}
&fib :: Integer \to Integer \\
&fib\ 0 = 0 \\
&fib\ 1 = 1 \\
&fib\ n = fib\ (n-1) + fib\ (n-2)
\end{aligned}
$$

- Polinomios de Hermite



$$
\begin{aligned}
&he\ 0\ x = 1 \\
&he\ 1\ x = 2 * x \\
&he\ n\ x = 2 * x * he\ (n-1)\ x - 2 * (n-1) * he\ (n-2)\ x
\end{aligned}
$$

Recursión general: división y resto

$$
\begin{aligned}
&\mathit{division} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}\\
&\mathit{division}\ m\ n\\
&\quad \mid m < n \qquad\quad = 0\\
&\quad \mid \mathit{otherwise} = 1 + \mathit{division}\ (m - n)\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{resto} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}\\
&\mathit{resto}\ m\ n\\
&\quad \mid m < n \qquad\quad = m\\
&\quad \mid \mathit{otherwise} = \mathit{resto}\ (m - n)\ n
\end{aligned}
$$

Recursión general: máximo común divisor

Usando el algoritmo de Euclides:

$$
\begin{array}{l}
mcd :: Integer \rightarrow Integer \rightarrow Integer \\
mcd\ m\ n \\
\quad \mid m == n = m \\
\quad \mid m > n \ \ = mcd\ (m - n)\ n \\
\quad \mid m < n \ \ = mcd\ m\ (n - m)
\end{array}
$$

Recursión general: máximo común divisor

Usando el algoritmo de Euclides:

```
mcd :: Integer -> Integer -> Integer
mcd m n
    | m == n = m
    | m > n  = mcd (m - n) n
    | m < n  = mcd m (n - m)
```

En forma equivalente (y mas eficiente):

```
fastmcd :: Integer -> Integer -> Integer
fastmcd m 0 = m
fastmcd m n = fastmcd n (m `mod` n)
```

Recursión general: McCarthy 91

La siguiente función, conocida como la función 91 de McCarthy, fue introducida en los años 70 por John McCarthy (el creador de LISP).

Es una definición en la que se usa recursión anidada.

$$
\begin{array}{l}
\mathit{mcCarthy91} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{mcCarthy91}\ n \mid n \leqslant 100 = \mathit{mcCarthy91}\ (\mathit{mcCarthy91}\ (n + 11)) \\
\qquad\qquad\qquad \mid n > 100\ = n - 10
\end{array}
$$

Recursión general: McCarthy 91

La siguiente función, conocida como la función 91 de McCarthy, fue introducida en los años 70 por John McCarthy (el creador de LISP).

Es una definición en la que se usa recursión anidada.

$$
\begin{aligned}
&\mathit{mcCarthy91} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{mcCarthy91}\ n \mid n \leqslant 100 = \mathit{mcCarthy91}\ (\mathit{mcCarthy91}\ (n + 11)) \\
&\qquad\qquad\qquad\ \ \mid n > 100\ \ = n - 10
\end{aligned}
$$

Lo interesante de esta función es su comportamiento:

$$
\begin{aligned}
&\mathit{mcCarthy91}\ n \mid n \leqslant 101 = 91 \\
&\qquad\qquad\qquad\ \ \mid n > 101\ \ = n - 10
\end{aligned}
$$

Recursión mutua: par-impar

$$
\begin{aligned}
&par :: Integer \to Bool\\
&par\ 0 = True\\
&par\ n = impar\ (n-1)
\end{aligned}
$$

$$
\begin{aligned}
&impar :: Integer \to Bool\\
&impar\ 0 = False\\
&impar\ n = par\ (n-1)
\end{aligned}
$$

Recursión sobre listas

Recursión estructural sobre listas

- Largo de una lista

$$
\begin{aligned}
&\mathit{length} :: [a] \rightarrow \mathit{Int} \\
&\mathit{length}\ [\,] \qquad\quad = 0 \\
&\mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs
\end{aligned}
$$

Recursión estructural sobre listas

- Largo de una lista

$$
\begin{aligned}
&length :: [a] \to Int\\
&length\ [\,] \qquad = 0\\
&length\ (x : xs) = 1 + length\ xs
\end{aligned}
$$

- Sumatoria de valores de una lista:

$$
\begin{aligned}
&sum :: Num\ a \Rightarrow [a] \to a\\
&sum\ [\,] \qquad = 0\\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

Recursión estructural sobre listas

- Largo de una lista

$$
\begin{aligned}
&length :: [a] \rightarrow Int \\
&length\ [\,] \qquad\quad = 0 \\
&length\ (x : xs) = 1 + length\ xs
\end{aligned}
$$

- Sumatoria de valores de una lista:

$$
\begin{aligned}
&sum :: Num\ a \Rightarrow [a] \rightarrow a \\
&sum\ [\,] \qquad\quad = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

- All

$$
\begin{aligned}
&all :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool \\
&all\ p\ [\,] \qquad\quad = True \\
&all\ p\ (x : xs) = p\ x\ \&\&\ all\ p\ xs
\end{aligned}
$$

map y filter

$$
\begin{aligned}
&map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]\\
&map\ f\ [\,] \qquad\quad = [\,]\\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

map y filter

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] \qquad\quad = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

$$
\begin{aligned}
&filter :: (a \to Bool) \to [a] \to [a] \\
&filter\ p\ [\,] \qquad\qquad\qquad\quad = [\,] \\
&filter\ p\ (x : xs) \mid p\ x \qquad\quad = x : filter\ p\ xs \\
&\qquad\qquad\qquad\ \mid otherwise = filter\ p\ xs
\end{aligned}
$$

Append, concat y reverse

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] \qquad\quad +\!\!+\ ys = ys \\
&(x : xs) +\!\!+\ ys = x : (xs +\!\!+\ ys)
\end{aligned}
$$

Append, concat y reverse

$$
\begin{aligned}
&(++) :: [a] \to [a] \to [a] \\
&[\,] \qquad\quad ++\ ys = ys \\
&(x : xs) ++\ ys = x : (xs ++ ys)
\end{aligned}
$$

$$
\begin{aligned}
&concat :: [[a]] \to [a] \\
&concat\ [\,] \qquad\quad = [\,] \\
&concat\ (xs : xss) = xs ++ concat\ xss
\end{aligned}
$$

Append, concat y reverse

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] \qquad\quad +\!\!+\; ys = ys \\
&(x : xs) +\!\!+\; ys = x : (xs +\!\!+\; ys)
\end{aligned}
$$

$$
\begin{aligned}
&concat :: [[a]] \to [a] \\
&concat\ [\,] \qquad\quad = [\,] \\
&concat\ (xs : xss) = xs +\!\!+\; concat\ xss
\end{aligned}
$$

$$
\begin{aligned}
&reverse :: [a] \to [a] \\
&reverse\ [\,] \qquad\ = [\,] \\
&reverse\ (x : xs) = reverse\ xs +\!\!+\; [x]
\end{aligned}
$$

insertion sort

$$
\begin{aligned}
&isort :: Ord\ a \Rightarrow [a] \rightarrow [a] \\
&isort\ [\,] \qquad\quad = [\,] \\
&isort\ (x : xs) = insert\ x\ (isort\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&insert :: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
&insert\ x\ [\,] \qquad\qquad\qquad\quad = [x] \\
&insert\ x\ (y : ys)\ \mid x \leqslant y \qquad = x : y : ys \\
&\qquad\qquad\qquad\quad \mid otherwise = y : insert\ x\ ys
\end{aligned}
$$

foldr

Las funciones anteriores pueden ser definidas en términos de *foldr*.

Recordemos que, dada una función f y un valor e,

$$foldr\ f\ e\ (x_1 : x_2 : \ldots : x_n : [])$$

retorna

$$x_1 \text{ `}f\text{` } (x_2 \text{ `}f\text{` } (\ldots(x_n \text{ `}f\text{` } e)\ldots))$$

Definición de *foldr*

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$
$$foldr\ f\ e\ []\ \ \ \ \ \ \ = e$$
$$foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)$$

Definición de *foldr*

$$\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ e\ [\,] \qquad\quad = e \\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}$$

- Es un esquema o patrón de recursión que captura definiciones por recursión estructural.
- Cualquier definición recursiva estructural de la forma:

$$\begin{aligned}
&h :: [a] \to b \\
&h\ [\,] \qquad\quad = e \\
&h\ (x : xs) = f\ x\ (h\ xs)
\end{aligned}$$

  se puede escribir como $h = foldr\ f\ e$.

- Un esquema de recursión similar se puede definir para cada tipo de dato algebraico.

Ejemplos

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

Ejemplos

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

$$sum = foldr\ (+)\ 0$$

Ejemplos

$$
\begin{aligned}
sum\ [\,] &= 0 \\
sum\ (x : xs) &= x + sum\ xs
\end{aligned}
$$

$$sum = foldr\ (+)\ 0$$

$$
\begin{aligned}
length\ [\,] &= 0 \\
length\ (x : xs) &= 1 + length\ xs
\end{aligned}
$$

Ejemplos

$$
\begin{aligned}
sum\ [\,] &= 0 \\
sum\ (x : xs) &= x + sum\ xs
\end{aligned}
$$

$$
sum = foldr\ (+)\ 0
$$

$$
\begin{aligned}
length\ [\,] &= 0 \\
length\ (x : xs) &= 1 + length\ xs
\end{aligned}
$$

$$
\begin{aligned}
&length = foldr\ flen\ 0 \\
&\quad \textbf{where}\ flen\ _\ r = 1 + r
\end{aligned}
$$

Ejemplos

$$
\begin{aligned}
&sum\ [\,] &&= 0\\
&sum\ (x : xs) &&= x + sum\ xs
\end{aligned}
\qquad
sum = foldr\ (+)\ 0
$$

$$
\begin{aligned}
&length\ [\,] &&= 0\\
&length\ (x : xs) &&= 1 + length\ xs
\end{aligned}
\qquad
\begin{aligned}
&length = foldr\ flen\ 0\\
&\quad \mathbf{where}\ flen\ _\ r = 1 + r
\end{aligned}
$$

$$
\begin{aligned}
&all\ p\ [\,] &&= True\\
&all\ p\ (x : xs) &&= p\ x\ \&\&\ all\ p\ xs
\end{aligned}
$$

Ejemplos

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

$$sum = foldr\ (+)\ 0$$

$$\begin{aligned} length\ [\,] &= 0 \\ length\ (x : xs) &= 1 + length\ xs \end{aligned}$$

$$\begin{aligned} & length = foldr\ flen\ 0 \\ & \quad \textbf{where}\ flen\ _\ r = 1 + r \end{aligned}$$

$$\begin{aligned} all\ p\ [\,] &= True \\ all\ p\ (x : xs) &= p\ x\ \&\&\ all\ p\ xs \end{aligned}$$

$$\begin{aligned} & all\ p = foldr\ fall\ True \\ & \quad \textbf{where}\ fall\ x\ r = p\ x\ \&\&\ r \end{aligned}$$

Ejemplos

$$concat = foldr\ (+\!\!+)\ [\,]$$

$$\begin{array}{l} reverse = foldr\ snoc\ [\,] \\ \quad \mathbf{where}\ snoc\ x\ r = r +\!\!+ [x] \end{array}$$

$$xs +\!\!+ ys = foldr\ (:)\ ys\ xs$$

$$map\ f = foldr\ ((:)\ .\ f)\ [\,]$$

$$\begin{array}{l} filter\ p = foldr\ fil\ [\,] \\ \quad \mathbf{where}\ fil\ x\ r \mid p\ x \qquad\quad = x : r \\ \qquad\qquad\qquad\ \mid otherwise = r \end{array}$$

$$isort = foldr\ insert\ [\,]$$

Funciones tail-recursivas

Las siguientes funciones nuevamente hacen recursión de manera estructural pero ahora con el agregado de que son tail-recursivas.

- Suma acumulativa:

$$
\begin{aligned}
& \mathit{sum} :: \mathit{Num}\ a \Rightarrow [a] \to a \\
& \mathit{sum} = \mathit{sumacc}\ 0 \\
& \quad \textbf{where}\ \mathit{sumacc}\ s\ [\,] \qquad\quad = s \\
& \qquad\qquad\quad \mathit{sumacc}\ s\ (x : \mathit{xs}) = \mathit{sumacc}\ (s + x)\ \mathit{xs}
\end{aligned}
$$

Funciones tail-recursivas

Las siguientes funciones nuevamente hacen recursión de manera estructural pero ahora con el agregado de que son tail-recursivas.

- Suma acumulativa:

$$
\begin{aligned}
&sum :: Num\ a \Rightarrow [a] \to a \\
&sum = sumacc\ 0 \\
&\quad \textbf{where}\ sumacc\ s\ [\,] \qquad\quad = s \\
&\quad\phantom{\textbf{where}}\ sumacc\ s\ (x : xs) = sumacc\ (s + x)\ xs
\end{aligned}
$$

- Reverse rápido:

$$
\begin{aligned}
&fastrev :: [a] \to [a] \\
&fastrev = revacc\ [\,] \\
&\quad \textbf{where}\ revacc\ r\ [\,] \qquad\quad = r \\
&\quad\phantom{\textbf{where}}\ revacc\ r\ (x : xs) = revacc\ (x : r)\ xs
\end{aligned}
$$

foldl

Funciones como las dos anteriores, tail-recursivas definidas por recursión estructural, son capturdas por una función llamada *foldl*.

$$foldl :: (b \to a \to b) \to b \to [a] \to b$$

Dada una función f y un valor v,

$$foldl\ f\ v\ (x_1 : x_2 : ... : x_n : [\,])$$

retorna

$$(...((v \text{ `}f\text{` } x_1) \text{ `}f\text{` } x_2)...) \text{ `}f\text{` } x_n$$

Definición de *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b\\
&\mathit{foldl}\ f\ \ v\ [\,] \qquad\quad = v\\
&\mathit{foldl}\ f\ \ v\ (x : xs) = \mathit{foldl}\ f\ \ (f\ \ v\ x)\ xs
\end{aligned}
$$

Definición de *foldl*

$$
\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b\\
&foldl\ f\ v\ [\,] \qquad\quad = v\\
&foldl\ f\ v\ (x : xs) = foldl\ f\ (f\ v\ x)\ xs
\end{aligned}
$$

- Estamos ante un nuevo esquema de recursión que ahora captura definiciones recursivas de la forma:

$$
\begin{aligned}
&h :: b \to [a] \to b\\
&h\ v\ [\,] \qquad\quad = v\\
&h\ v\ (x : xs) = h\ (f\ v\ x)\ xs
\end{aligned}
$$

- Notar que el valor v juega el rol de acumulador, al que hay que darle un valor inicial.

Ejemplos

- Suma acumulativa:

$$sum :: Num\ a \Rightarrow [a] \rightarrow a$$
$$sum = sumacc\ 0$$
$$\quad \textbf{where}\ sumacc\ s\ [\,] \qquad = s$$
$$\qquad\qquad sumacc\ s\ (x : xs) = sumacc\ (s + x)\ xs$$

$$sum = foldl\ (+)\ 0$$

Ejemplos

- Suma acumulativa:

$$
\begin{aligned}
&sum :: Num\ a \Rightarrow [a] \rightarrow a\\
&sum = sumacc\ 0\\
&\quad \textbf{where}\ sumacc\ s\ [\,] \qquad\quad = s\\
&\qquad\qquad\ \ sumacc\ s\ (x : xs) = sumacc\ (s + x)\ xs
\end{aligned}
$$

$$sum = foldl\ (+)\ 0$$

- Reverse rápido:

$$
\begin{aligned}
&fastrev :: [a] \rightarrow [a]\\
&fastrev = revacc\ [\,]\\
&\quad \textbf{where}\ revacc\ r\ [\,] \qquad\quad = r\\
&\qquad\qquad\ \ revacc\ r\ (x : xs) = revacc\ (x : r)\ xs
\end{aligned}
$$

$$fastrev = foldl\ (flip\ (:))\ [\,]$$

init y last

$$
\begin{aligned}
&init :: [a] \to [a] \\
&init\ [x] \qquad = [\,] \\
&init\ (x : xs) = x : init\ xs
\end{aligned}
$$

$$
\begin{aligned}
&last :: [a] \to a \\
&last\ [x] \qquad = x \\
&last\ (x : xs) = last\ xs
\end{aligned}
$$

Recursión sobre múltiples argumentos

$$
\begin{array}{llll}
zip :: [a] \to [b] \to [(a,b)] \\
zip\ [\,] & _ & = [\,] \\
zip\ _ & [\,] & = [\,] \\
zip\ (x : xs)\ (y : ys) & & = (x, y) : zip\ xs\ ys
\end{array}
$$

$$
\begin{array}{llll}
zipWith :: (a \to b \to c) \to [a] \to [b] \to [c] \\
zipWith\ f\ [\,] & _ & = [\,] \\
zipWith\ f\ _ & [\,] & = [\,] \\
zipWith\ f\ (x : xs)\ (y : ys) & & = f\ x\ y : zipWith\ f\ xs\ ys
\end{array}
$$

Recursión sobre múltiples argumentos

$$
\begin{aligned}
&\mathit{take} :: \mathit{Int} \to [a] \to [a] \\
&\mathit{take}\ n\ _ \mid n \leqslant 0 = [\,] \\
&\mathit{take}\ _\ [\,] \qquad\qquad = [\,] \\
&\mathit{take}\ n\ (x : \mathit{xs}) \quad = x : \mathit{take}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{drop} :: \mathit{Int} \to [a] \to [a] \\
&\mathit{drop}\ n\ \mathit{xs} \mid n \leqslant 0 = \mathit{xs} \\
&\mathit{drop}\ _\ [\,] \qquad\qquad = [\,] \\
&\mathit{drop}\ n\ (_ : \mathit{xs}) \quad = \mathit{drop}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{splitAt} :: \mathit{Int} \to [a] \to ([a],[a]) \\
&\mathit{splitAt}\ n\ \mathit{xs} \mid n \leqslant 0 = ([\,], \mathit{xs}) \\
&\mathit{splitAt}\ _\ [\,] \qquad\qquad = ([\,],[\,]) \\
&\mathit{splitAt}\ n\ (x : \mathit{xs}) \quad = (x : \mathit{ys}, \mathit{zs}) \\
&\quad \textbf{where}\ (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

takeWhile y *dropWhile*

$$
\begin{aligned}
&takeWhile :: (a \to Bool) \to [a] \to [a]\\
&takeWhile\ p\ [\,] && = [\,]\\
&takeWhile\ p\ (x : xs) \mid p\ x && = x : takeWhile\ p\ xs\\
&\qquad\qquad\qquad\quad\ \mid otherwise && = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&dropWhile :: (a \to Bool) \to [a] \to [a]\\
&dropWhile\ p\ [\,] && = [\,]\\
&dropWhile\ p\ ys@(x : xs) \mid p\ x && = dropWhile\ p\ xs\\
&\qquad\qquad\qquad\qquad\quad\ \mid otherwise && = ys
\end{aligned}
$$

Recursión general: quicksort

$$
\begin{aligned}
&qsort :: Ord\ a \Rightarrow [a] \rightarrow [a] \\
&qsort\ [\,] = [\,] \\
&qsort\ (x : xs) = qsort\ leq_x +\!\!+ [x] +\!\!+ qsort\ gt_x \\
&\quad \textbf{where}\ leq_x \quad = [y \mid y \leftarrow xs, y \leqslant x] \\
&\qquad\qquad gt_x \quad = [y \mid y \leftarrow xs, y > x]
\end{aligned}
$$