GLOSARIO

```
curry :: ((a, b) -> c) -> a -> b -> c -- Devuelve una función currificada
curry f x y = f (x, y)
uncurry :: (a -> b -> c) -> (a, b) -> c -- Devuelve una función no currificada
uncurry f (x, y) = f x y
```

Folds

```
foldr :: (a -> b -> b) -> b -> [a] -> b  -- Fold derecho (asociativo a derecha).
foldl :: (b -> a -> b) -> b -> [a] -> b  -- Fold izquierdo (asociativo a izquierda).
foldr1 :: (a -> a -> a) -> [a] -> a      -- `foldr` en listas no vacías.
foldl1 :: (a -> a -> a) -> [a] -> a      -- `foldl` en listas no vacías.
```

Listas

```
map :: (a -> b) -> [a] -> [b]                    -- Aplica función a cada elemento.
filter :: (a -> Bool) -> [a] -> [a]    -- Filtra elementos que cumplen un predicado.
concat :: [[a]] -> [a]                           -- Concatena listas anidadas.
concatMap :: (a -> [b]) -> [a] -> [b]            -- `concat . map`.
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]    -- Combina listas con una función.
reverse :: [a] -> [a]                            -- Invierte la lista.
```

Cosas sobre listas

```
(++) :: [a] -> [a] -> [a]      -- Concatena listas.
head :: [a] -> a               -- Primer elemento (error si vacía).
tail :: [a] -> [a]             -- Lista sin el primer elemento.
init :: [a] -> [a]             -- Lista sin el último elemento.
last :: [a] -> a               -- Último elemento (error si vacía).
length :: [a] -> Int           -- Longitud de la lista.
null :: [a] -> Bool            -- Verifica si la lista está vacía.
replicate :: Int -> a -> [a]   -- Repite un elemento `n` veces.
take :: Int -> [a] -> [a]      -- Toma los primeros `n` elementos.
drop :: Int -> [a] -> [a]      -- Elimina los primeros `n` elementos.
```

Búsqueda

```
elem :: Eq a => a -> [a] -> Bool   -- Verifica si un elemento está en la lista.
any :: (a -> Bool) -> [a] -> Bool  -- Verifica si algún elemento cumple el predicado.
all :: (a -> Bool) -> [a] -> Bool  -- Verifica si todos cumplen el predicado.
```

Orden y duplicados

```
sort :: Ord a => [a] -> [a]                  -- Ordena una lista.
sortBy :: (a -> a -> Ordering) -> [a] -> [a] -- Ordena con función de comparación.
nub :: Eq a => [a] -> [a]                    -- Elimina elementos duplicados.
union :: Eq a => [a] -> [a] -> [a]           -- Unión de conjuntos (sin duplicados).
intersect :: Eq a => [a] -> [a] -> [a]       -- Intersección de conjuntos.
```

Aritmética, lógica y comparación

```
sum :: Num a => [a] -> a                          -- Suma de elementos.
mod :: Integral a => a -> a -> a                  -- Módulo.
div :: Integral a => a -> a -> a                  -- División entera.
odd :: Integral a => a -> Bool                    -- Verifica si es impar.
even :: Integral a => a -> Bool                   -- Verifica si es par.
and :: [Bool] -> Bool                             -- AND lógico sobre una lista.
or :: [Bool] -> Bool                              -- OR lógico sobre una lista.
not :: Bool -> Bool                               -- Negación lógica.
(==), (/=) :: Eq a => a -> a -> Bool              -- Igualdad/desigualdad.
compare :: Ord a => a -> a -> Ordering            -- Compara (`LT`, `EQ`, `GT`).
comparing :: Ord a => (b -> a) -> b -> b -> Ordering -- Compara usando una función.
max, min :: Ord a => a -> a -> a                  -- Máximo/mínimo entre dos valores.
maximum, minimum :: Ord a => [a] -> a             -- Máximo/mínimo de una lista.
maximumBy, minimumBy :: (a -> a -> Ordering) -> [a] -> a -- Máximo/mínimo con comparador.
```

Esquemas de recursión:

- **Estructural:** permite acceder a los argumentos no recursivos de los constructores, y a los resultados de la recursión para las subestructuras.
- **Primitiva:** como la estructural, pero además permite acceder a las subestructuras.
- **Global:** como la primitiva, pero además permite acceder a los resultados de las recursiones anteriores

Ejemplos:

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs

insertarOrdenado e [] = [e]
insertarOrdenado e (x:xs) = if e < x then e:x:xs
                                     else x:(insertarOrdenado e xs)

elementosEnPosicionesPares [] = []
elementosEnPosicionesPares (x:xs) = if null xs then [x]
                                  else x:elementosEnPosicionesPares (tail xs)
```

1. La recursión de `longitud` es **estructural**, porque hace recursión sobre la cola de la lista (xs) pero no accede a la cola en sí, ni a resultados de recursiones anteriores.
2. La recursión de `insertarOrdenado` es **primitiva** porque accede directamente a xs (además de hacer recursión), pero no accede a los resultados anteriores.
3. La recursión de `elementosEnPosicionesPares` es **global**, ya que accede a un resultado anterior: el de la recursión sobre la cola de la cola de la lista (es decir tail xs).

Funciones clave:

```
foldr :: (a -> b -> b) -> b -> [a] -> b -- Estructural
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (b -> a -> b) -> b -> [a] -> b -- Estructural
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

recr :: (a -> [a] -> b -> b) -> b -> [a] -> b -- Primitiva
recr _ z []     = z
recr f z (x:xs) = f x xs (recr f z xs)
```

Construcción de folds personalizados:

Un esquema de recursión estructural espera recibir un argumento por cada constructor (para saber qué devolver en cada caso), y además la estructura que va a recorrer.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

Un pequeño ejemplo:

```
data AB a = Hoja | Nodo (AB a) a (AB a)
  -- Constructor base: Hoja
  -- Constructor recursivo: Nodo

foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB casoHoja _ Hoja = casoHoja  -- Para Hoja
foldAB casoHoja f (Nodo izq val der) =
   f (foldAB casoHoja f izq) val (foldAB casoHoja f der)
   -- Para Nodo: recibe resultados recursivos (izq/der) y el valor 'val'
```

Extensionalidad:

```
Dadas f, g :: a -> b, probar f = g se reduce a probar:
                              ∀ x::a . f x = g x
```

Propiedades útiles:

```
∀F :: a -> b . ∀G :: a -> b . ∀Y :: b . ∀Z :: a .
```

1. `F = G` $\Leftrightarrow$ `∀x::a. F x = G x`
2. `F = \x -> Y` $\Leftrightarrow$ `∀x::a. F x = Y`
3. `(x -> Y) Z` $=_{\beta}$ `Y` reemplazando `x` por `Z`
4. `\x -> F x` $=_{\eta}$ `F`

F, G, Y y Z pueden ser expresiones complejas, siempre que `x` no aparezca libre en F, G, ni Z

Ejemplos

2. `F x = x * 2` y `G = \x -> x * 2` => `F = G`
3. `(\x -> x + 5) 3 =β (3 + 5)`

Lemas de generación

Para pares: `Si p :: (a, b), entonces ∃x :: a. ∃y :: b. p = (x, y).`

Para sumas `data Either a b = Left a | Right b`:

```
Si e :: Either a b, entonces:
  o bien ∃x :: a. e = Left x
  o bien ∃y :: b. e = Right y
```

Idea general para una demostración por inducción estructural

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.
- Plantear el esquema de inducción.
- Plantear y resolver el o los caso(s) base.
- Plantear y resolver el o los caso(s) inductivo(s).

Dato: Los argumentos no recursivos quedan cuantificados universalmente.

Ejemplo: principio de inducción sobre listas

`data [a] = [] | a : [a]`

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `[a]` tal que:

- $\mathcal{P}(\texttt{[]})$
- $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\underbrace{\mathcal{P}(\texttt{xs})}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{x : xs})}_{\text{T.I.}})$

Entonces $\forall \texttt{xs} :: \texttt{[a]}.\ \mathcal{P}(\texttt{xs})$.

Ejemplo: principio de inducción sobre árboles binarios

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

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `AB a` tal que:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{i} :: \texttt{AB a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{d} :: \texttt{AB a}.$

$$(\underbrace{(\mathcal{P}(\texttt{i}) \wedge \mathcal{P}(\texttt{d}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin i r d})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{AB a}.\ \mathcal{P}(\texttt{x})$.

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `Poli a` tal que:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \texttt{a}.\ \mathcal{P}(\texttt{Cte k})$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$

$$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suma p q})}_{\text{T.I.}})$$

- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$

$$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{Poli a}.\ \mathcal{P}(\texttt{x})$.

Si no podemos continuar en un paso de la demostración, optamos por demostrar un lema sobre la operación.

Lógica intuicionista

Reglas básicas:

$$\frac{}{\Gamma, \tau \vdash \tau}\, ax$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau \wedge \sigma}\, \wedge_i$$

$$\frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \tau}\, \wedge_{e_1} \qquad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \sigma}\, \wedge_{e_2}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \vee \sigma}\, \vee_{i_1} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \tau \vee \sigma}\, \vee_{i_2}$$

$$\frac{\Gamma \vdash \tau \vee \sigma \qquad \Gamma, \tau \vdash \rho \qquad \Gamma, \sigma \vdash \rho}{\Gamma \vdash \rho}\, \vee_e$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \tau}\, \bot_e$$

$$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \Rightarrow \sigma}\, \Rightarrow_i$$

$$\frac{\Gamma \vdash \tau \Rightarrow \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma}\, \Rightarrow_e$$

$$\frac{\Gamma, \tau \vdash \bot}{\Gamma \vdash \neg\tau}\, \neg_i$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \neg\tau}{\Gamma \vdash \bot}\, \neg_e$$

Reglas derivadas:

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \neg\neg\tau}\, \neg\neg_i \qquad \frac{\Gamma \vdash \tau \Rightarrow \sigma \qquad \Gamma \vdash \neg\sigma}{\Gamma \vdash \neg\tau}\, MT$$

Lógica clásica

Reglas básicas:

$$\frac{\Gamma \vdash \neg\neg\tau}{\Gamma \vdash \tau}\, \neg\neg_e$$

Reglas derivadas:

$$\frac{\Gamma, \neg\tau \vdash \bot}{\Gamma \vdash \tau}\, PBC \qquad \frac{}{\Gamma \vdash \tau \vee \neg\tau}\, LEM$$

Sintaxis y tipado

Cosas a tener en cuenta

Asumimos que la aplicación es asociativa a izquierda:

$$M\,N\,P \quad = \quad (M\,N)\,P \quad \neq \quad M\,(N\,P)$$

La abstracción y el "if" tienen menor precedencia que la aplicación:

$$\lambda x : \tau.\, M\,N \quad = \quad \lambda x : \tau.\, (M\,N) \quad \neq \quad (\lambda x : \tau.\, M)\,N$$

Tipos y términos

Las expresiones de tipos (o simplemente tipos) son

$$\sigma ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \sigma \to \sigma$$

Sea $\mathcal{X}$ un conjunto infinito enumerable de variables y $x \in \mathcal{X}$. Los términos están dados por

$$\begin{aligned} M ::= {} & x \\ & \mid \lambda x : \sigma.M \\ & \mid M\ M \\ & \mid \mathsf{true} \\ & \mid \mathsf{false} \\ & \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \\ & \mid \mathsf{zero} \\ & \mid \mathsf{succ}(M) \\ & \mid \mathsf{pred}(M) \\ & \mid \mathsf{isZero}(M) \end{aligned}$$

Axiomas y reglas de tipado

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \textit{T-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \textit{T-False}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ \textit{T-Var}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool} \quad \Gamma \vdash P : \sigma \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : \sigma}\ \textit{T-If}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}\ \textit{T-Abs} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ \textit{T-App}$$

$$\frac{}{\Gamma \vdash \text{zero} : \text{Nat}}\ \textit{T-Zero}$$

$$\frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{succ}(M) : \text{Nat}}\ \textit{T-Succ} \qquad \frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{pred}(M) : \text{Nat}}\ \textit{T-Pred}$$

$$\frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{isZero}(M) : \text{Bool}}\ \textit{T-IsZero}$$

Semántica operacional

$$V ::= \text{true} \mid \text{false} \mid \lambda x : \sigma.M \mid \text{zero} \mid \text{succ}(V)$$

(Los valores de tipo Nat pueden escribirse como $\underline{n}$, lo cual abrevia $\text{succ}^n(\text{zero})$).

Reglas de evaluacion en un paso

Propiedades de la evaluación:

Teorema (Determinismo)

Si $M \to N_1$ y $M \to N_2$ entonces $N_1 = N_2$.

Teorema (Preservación de tipos)

Si $\vdash M : \tau$ y $M \to N$ entonces $\vdash N : \tau$.

Teorema (Progreso)

Si $\vdash M : \tau$ entonces:

1. O bien M es un valor.
2. O bien existe N tal que $M \to N$.

Teorema (Terminación)

Si $\vdash M : \tau$, entonces no hay una cadena infinita de pasos:

$$M \to M_1 \to M_2 \to \dots$$

Si $M_1 \to M_1'$, **entonces** $M_1 M_2 \to M_1' M_2$ $\qquad (\textit{E-App}_1 \text{ o } \mu)$

Si $M_2 \to M_2'$, **entonces** $V M_2 \to V M_2'$ $\qquad (\textit{E-App}_2 \text{ o } \nu)$

$$(\lambda x : \sigma.M)V \to M\{x := V\} \qquad (\textit{E-AppAbs o } \beta\ )$$

Ejemplos:

μ:

$$\underbrace{(\lambda x : \mathrm{Nat}.x)(\lambda y : \mathrm{Nat}.y)}_{M_1}\ \underbrace{3}_{M_2} \underset{\beta}{\to} \underbrace{(\lambda y : \mathrm{Nay}.y)}_{M_1\prime}\ \underbrace{3}_{M_2}$$

ν:

$$\underbrace{(\lambda x : \mathrm{Nat}.\mathrm{succ}(x))}_{V}\underbrace{((\lambda y : \mathrm{Nat}.y)\ 3)}_{M_2} \underset{\beta}{\to} \underbrace{(\lambda x : \mathrm{Nat}.\mathrm{succ}(x))}_{V}\ \underbrace{3}_{M_2\prime},$$

β:

$$(\lambda x : \mathrm{Nat}.\mathrm{succ}(x))\ 2 \to \mathrm{succ}(x)\{x := 2\} = \mathrm{succ}(2)$$

$$\text{if true then } M_2 \text{ else } M_3 \to M_2 \qquad (\textit{E-IfTrue})$$

$$\text{if false then } M_2 \text{ else } M_3 \to M_3 \qquad (\textit{E-IfFalse})$$

$$\textbf{Si } M_1 \to M_1', \textbf{ entonces}$$
$$\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \to \text{if } M_1' \text{ then } M_2 \text{ else } M_3 \qquad (\textit{E-If})$$

$$\text{pred}(\text{succ}(\underline{n})) \to \underline{n} \qquad (\textit{E-PredSucc})$$

$$\textbf{Opcional*: } \text{pred}(\text{zero}) \to \text{zero} \qquad (\textit{E-Pred}_0)$$

$$\text{isZero}(\text{zero}) \to \text{true} \qquad (\textit{E-IsZero}_0)$$

$$\text{isZero}(\text{succ}(\underline{n})) \to \text{false} \qquad (\textit{E-IsZero}_n)$$

$$\textbf{Si } M \to N, \textbf{ entonces } \text{succ}(M) \to \text{succ}(N) \qquad (\textit{E-Succ})$$

$$\textbf{Si } M \to N, \textbf{ entonces } \text{pred}(M) \to \text{pred}(N) \qquad (\textit{E-Pred})$$

$$\textbf{Si } M \to N, \textbf{ entonces } \text{isZero}(M) \to \text{isZero}(N) \qquad (\textit{E-IsZero})$$

Forma normal ("f.n.")

Un programa M es una **f.n.** si no existe M' tal que $M \to M'$.

Podemos también expresar macros, como:

$$\mathrm{curry}_{\sigma,\tau,\delta} = \lambda f : \sigma \times \tau \to \delta.\lambda x : \sigma.\lambda y : \tau.f\langle x, y\rangle$$

IMPORTANTE:

A la hora de hacer reglas de tipado, hay que hacer una para cada expresión nueva.

Luego hay que extender el conjunto de valores.

Hay que hacer reglas de congruencia que mantengan determinismo (ir reduciendo congruentemente parte por parte)

Hay que hacer axiomas para cada valor nuevo del conjunto de valores