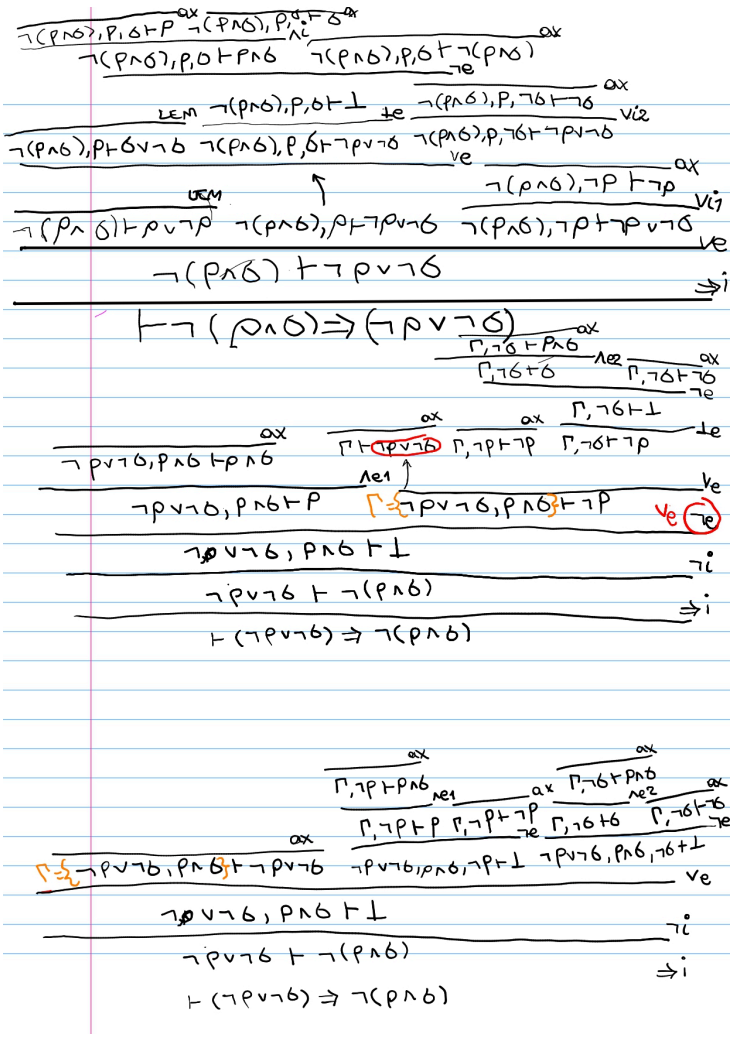
```haskell
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \x -> \y -> f (x, y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f = \(x, y) -> f x y

misum :: [Float] -> Float
misum s = foldr (\acc x -> acc + x) 0 s

mielem :: Eq a => a -> [a] -> Bool
mielem e s = foldr (\x acc -> acc || e==x) False s

mimasmas :: [a] -> [a] -> [a]
mimasmas s1 s2 = foldr(\x acc -> x:acc) s2 s1

mifilter :: (a -> Bool) -> [a] -> [a]
mifilter f s = foldr(\x acc -> if f x  then x:acc else acc)
      [] s

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

recAB ::
   b                                  -- Nil
   -> (AB a -> a -> AB a -> b -> b -> b)   -- Bin
   -> AB a
   -> b

recAB z f x = case x of
   Nil -> z
   (Bin l v r) -> f l v r (rec l) (rec r)
   where rec = recAB z f

foldAB ::
   b                    -- Nil
   -> (b -> a -> b -> b)   -- Bin
   -> AB a
   -> b

foldAB z f x = case x of
   Nil -> z
   (Bin l v r) -> f (rec l) v (rec r)
   where rec = foldAB z f

foldAB :: (a -> b -> b -> b) -> b -> AB a -> b
foldAB f1 f2 = recAB (\a _ _ -> f1 a) f2


      esNil :: AB a -> Bool
      esNil x = case x of
         Nil -> True
         _ -> False

cantNodos :: AB a -> Int
cantNodos = foldAB (const 1) (\ri _ rd -> 1 + ri + rd)

sumatoriaArbol :: AB Int -> Int
sumatoriaArbol = foldAB id (\left r right -> left + r + right)

mejorSegún :: (a -> a -> Bool) -> AB a -> a
mejorSegún f (Bin l v r) = foldAB v (\rl v rr -> (rl `g` v) `g` rr)
(Bin l v r)
   where g x y = if f x y then x else y

esABB :: Ord a => AB a -> Bool
esABB = recAB True (\ l v r rl rr -> (esNil l || mejorSegún (>) l
<= v) && (esNil r || mejorSegún (<) r >= v) && rl && rr)

raíz :: AB a -> a
raíz (Bin l v r) = v

data RoseTree a = Rose a [RoseTree a]

tamaño :: RoseTree a -> Int
tamaño (Rose x hijos) = 1 + sum (map tamaño hijos)
-- tamaño = foldRT (\_ recs -> 1 + sum recs)

foldRT :: (a -> [b] -> b) -> RoseTree a -> b
foldRT f (Rose x hijos) = f x (map (foldRT f) hijos)

data Polinomio a = X
            | Cte a
            | Suma (Polinomio a) (Polinomio a)
            | Prod (Polinomio a) (Polinomio a)

evaluar n poli = case poli of
      X -> n
      Cte k -> k
      Suma p q -> evaluar n p + evaluar n q
      Prod p q -> evaluar n p * evaluar n q


-- flip
flip' :: (a -> b -> c) -> b -> a -> c
-- flip' f y x = f x y
flip' f = \y x -> f x y
```

```
data Componente
= Contenedor
  | Motor
  | Escudo
  | Cañon
  deriving Eq

data NaveEspacial
= Modulo Componente NaveEspacial NaveEspacial
| Base Componente
deriving Eq

recNave :: ( Componente -> NaveEspacial ->
NaveEspacial -> a -> a -> a)
-> ( Componente -> a ) -> NaveEspacial -> a

recNave f1 f2 n = case n of
        Modulo c n1 n2 -> f1 c n1 n2 ( rec n1 ) ( rec n2
)
        Base c -> f2 c
where rec = recNave f1 f2

foldNave :: ( Componente -> a -> a -> a )
-> ( Componente -> a ) -> NaveEspacial -> a

foldNave f1 f2 = recNave (\ c _ _ -> f1 c ) f2

espejo :: NaveEspacial -> NaveEspacial
espejo = foldNave (\ c r1 r2 -> Modulo c r2 r1 ) Base

esSubnavePropia :: NaveEspacial -> NaveEspacial ->
Bool
esSubnavePropia n1 =
recNave (\ _ sn1 sn2 r1 r2 -> sn1 == n1 ||
sn2 == n1 || r1 || r2 )
( const False )
```

Dada una nave y un numero natural n, devuelve una nave con los niveles 0 a n de la original, siendo el nivel 0 la ra´ız.

```
truncar :: NaveEspacial -> Integer -> NaveEspacial
truncar =
   foldNave (\ c r1 r2 -> \ i ->
         if i == 0
         then ( Base c )
         else ( M ´o dulo c ( r1 (i -1) ) ( r2 (i -1) ) ) )
      (\ c -> \ i -> Base c )
```



Se desea extender el Cálculo Lambda tipado con colas bidireccionales (también conocidas como *deque*).
Se extenderán los tipos y términos de la siguiente manera:

$\tau ::= \cdots \mid \text{Cola}_\tau$

$M ::= \cdots \mid \langle\rangle_\tau \mid M \bullet M \mid \text{próximo}(M) \mid \text{desencolar}(M)$
$\mid \text{case } M \text{ of } \langle\rangle \rightsquigarrow M; c \bullet x \rightsquigarrow M$

donde $\langle\rangle_\sigma$ es la cola vacía en la que se pueden encolar elementos de tipo σ; $M_1 \bullet M_2$ representa el agregado del elemento M_2 al **final** de la cola M_1; los observadores próximo(M_1) y desencolar(M_1) devuelven, respectivamente, el primer elemento de la cola (el primero que se encoló), y la cola sin el primer elemento (estos dos últimos solo tienen sentido si la cola no es vacía); y el observador case M_1 of $\langle\rangle \rightsquigarrow M_2; c \bullet x \rightsquigarrow M_3$ permite operar con la cola en sentido contrario, accediendo al último elemento encolado (cuyo valor se ligará a la variable x en M_3) y al resto de la cola (que se ligará a la variable c en el mismo subtérmino).

1. Introducir las reglas de tipado para la extensión propuesta.
2. Definir el conjunto de valores y las nuevas reglas de reducción. Pueden usar los conectivos booleanos de la guía. **No es necesario escribir las reglas de congruencia,** basta con indicar cuántas son.
   Pista: puede ser necesario mirar más de un nivel de un término para saber a qué reduce.
3. Mostrar paso por paso cómo reduce la expresión:
   case $\langle\rangle_{\text{Nat}} \bullet \underline{1} \bullet 0$ of $\langle\rangle \rightsquigarrow$ próximo($\langle\rangle_{\text{Bool}}$); $c \bullet x \rightsquigarrow$ isZero(x)
4. Definir como macro la función $\texttt{último}_\tau$, que dada una cola devuelve el último elemento que se encoló en ella. Si la cola es vacía, puede colgarse o llegar a una forma normal bien tipada que no sea un valor. Dar un juicio de tipado válido para esta función (no es necesario demostrarlo).