Práctica para Primer Parcial

Tomás Felipe Melli

May 12, 2025

Índice

1 1C2024 Parcial

1.1 Ejercicio 1

El siguiente tipo de datos sirve para representar árboles ternarios:

```
data AT a = NilT | Tri a (AT a) (AT a) (AT a)
```

Definimos el siguiente árbol para los ejemplos:

```
at1 = Tri 1 (Tri 2 NilT NilT NilT) (Tri 3 (Tri 4 NilT NilT NilT) NilT NilT) (Tri 5 NilT NilT NilT)
```

1. Para el primer punto nos piden definir **foldAT**

```
foldAT :: b -> ( a -> b -> b -> b -> b) -> AT a -> b
foldAT fNilT fTri t = case t of
        NilT -> fNilT
        Tri r izq med der -> fTri r (rec izq) (rec med) (rec der)
          where rec = foldAT fNilT fTri
```

   Explicación del tipo : la función *fNIl* retorna algo del tipo b. La función *fTri* toma el tipo de la raiz (:a) y sus otros 3 parámetros ya fueron procesados, por tanto son de tipo b. Como consecuencia se retorna algo del tipo b (resultado). El otro parámetro de *foldAT* es el árbol en sí.

2. Un recorrido *pre-order* es visitar la raiz, luego el subárbol izquierdo, (en este caso al medio luego) y finalmente, el derecho . Si nos piden un *post-order*(cuando devolvemos primero el izquierdo, luego el derecho y finalmente la raiz) o mismo un *inorder* (izquierdo, raiz, derecho) la idea es cambiar el orden de los parámetros que recibe *++*. (Esto en árboles binarios). Primero lo hacemos con recursión explícita :

```
preorder :: AT a -> [a]
preorder NilT = []
preorder (Tri r izq med der) = [r] ++ preorder izq ++ preorder med ++ preorder der
```

   Ahora que ya vemos cómo hacerlo, lo podemos llevar al **foldAT**

```
preorder :: AT a -> [a]
preorder t = foldAT [] (\r izq med der -> [r] ++ izq ++ med ++ der ) t
```

   La t del final la podemos sacar, queda más declarativo, así que la dejo.

3. La función **mapAT**. Misma idea, lo hacemos ...

```
mapAT :: (a -> b) -> AT a -> AT b
mapAT _ NilT = NilT
mapAt f (Tri r izq med der) = f r (mapAT izq) (mapAT med) (mapAT der)
```

```
mapAT :: (a -> b) -> AT a -> AT b
mapAT f t = foldAT NilT (\r izq med der -> Tri (f r) izq med der ) t
```

   La idea es como venimos viendo, queremos procesar el elemento en r, el resto ya viene resuelto de la recursión, así que simplemente nos construimos el árbol ternario aplicando f a la raiz y con todo lo que ya viene de la recursión.

4. Nos piden definir la función nivel que devuelve los nodos del nivel correspondiente del árbol (consideramos 0 al nivel de la raiz).

```
nivel :: AT a-> Int -> [a]
nivel NilT _  = []
nivel (Tri r _ _ _) 0  = [r]
nivel (Tri r izq med der) n  | n > 0 = nivel izq (n-1)  ++ nivel med (n-1)  ++ nivel der (n-1)
```

   Pensemos cómo llevar esto al foldAT

```
nivel :: AT a -> Int -> [a]
nivel t n = foldAT (const [])
        (\r izq med der i -> if i == 0 then [r] else ((izq (i-1)) ++ (med (i-1)) ++ (der (i-1)))) t n
```

   EL tema de esta función, como vimos en su forma de recursión explícita es el *(n-1)*. Necesitamos que la λ tenga acceso al decremento de ese n. Para ello, se la pasamos y listo. La idea es análoga, queremos que si el nivel es 0 devuelva root, sino, que me una los nodos que están en el mismo nivel. Por qué usamos (const []) ? No tipa sino, Haskell nos dice que necesitamos poner algo de tipo $Int \to [a]$. Pongo el t por declarativid

1.2 Ejercicio 2

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)

    const :: a -> b -> a
{C} const = \x -> \y -> x

    length :: [a] -> Int
{L0}length [] = 0
{L1}length (x:xs) = 1 + length xs

    head :: [a] -> a
{H} head (x:xs) = x

    null :: [a] -> Bool
{N0}null [] = True
{N1}null (x:xs) = False

    tail :: [a] -> [a]
{T} tail (x:xs) = xs

    altura :: AEB a -> Int
{A0}altura (Hoja x) = 1
{A1}altura (Bin i r d) = 1 + max (altura i) (altura d)

    esPreRama :: Eq a => AEB a -> [a] -> Bool
{E0}esPreRama (Hoja x) = \xs -> null xs || (xs == [x])
{E1}esPreRama (Bin i r d) =
    (\xs -> null xs || (r == head xs && (esPreRama i (tail xs) || esPreRama d (tail xs)))
```

1. Asumimos que $Eq\ a$ para demostrar

$$\forall t :: AEB\ a. \forall xs : [a].\quad esPreRama\ t\ xs \Rightarrow length\ xs \leq altura\ t$$

Para comenzar la demostración, primero vamos a decir que vamos a hacer inducción sobre el árbol, o sea que queremos probar que $\forall t :: AEB\ a. \forall xs : [a].\quad P(t) = esPreRama\ t\ xs \Rightarrow length\ xs \leq altura\ t$. Como se trata de una implicación, vamos a asumir que el antecedente vale. Dicho esto, comenzamos con la demo :

- Caso Base : qvq

$$\begin{aligned} P(Hoja\ x) &= esPreRama\ (Hoja\ x)\ xs \Rightarrow length\ xs \leq altura\ (Hoja\ x) \\ &\overset{\{A0\}}{\equiv} esPreRama\ (Hoja\ x)\ xs \Rightarrow length\ xs \leq 1 \end{aligned}$$

Dividimos en casos según la lista xs es vacía o no-vacía

– Caso xs vacía

$$\begin{aligned} & esPreRama\ (Hoja\ x)\ [\,] \Rightarrow length\ [\,] \leq 1 \\ &\overset{\{E0,\beta\}}{\equiv} null\ [\,] \mid\mid\ ([\,] == [x]) \Rightarrow length\ [\,] \leq 1 \\ &\overset{\{N0,L0\}}{\equiv} True \mid\mid\ False \Rightarrow 0 \leq 1 \\ &\equiv True \Rightarrow 0 \leq 1 \\ &\equiv True \end{aligned}$$

– Caso xs no-vacía

$$\begin{aligned} & esPreRama\ (Hoja\ x)\ xs \Rightarrow length\ xs \leq 1 \\ &\overset{\{E0\}}{\equiv} null\ xs \mid\mid\ (xs == [x]) \Rightarrow length\ xs \leq 1 \\ &\overset{\{N0\}}{\equiv} (xs == [x]) \Rightarrow length\ xs \leq 1 \end{aligned}$$

Tenemos otros dos casos, si xs efectivamente es $[x]$ o si no lo es.

* Caso $xs == [x]$

$$\equiv ([x] == [x]) \Rightarrow length\ (x : [\,]) \leq 1$$
$$\overset{\{L1\}}{\equiv} True \Rightarrow 1 + length[\,] \leq 1$$
$$\overset{\{L0\}}{\equiv} True \Rightarrow 1 + 0 \leq 1$$
$$\equiv True \Rightarrow 1 \leq 1$$
$$\equiv True \Rightarrow True$$
$$\equiv True$$

* Caso $xs \neq [x]$

$$\equiv (xs == [x]) \Rightarrow length\ xs \leq 1$$
$$\equiv False \Rightarrow length\ xs \leq 1$$
$$\equiv True$$

- Paso Inductivo: qvq $\forall i : AB\ a, r : a, d : AB\ a. \quad P(i) \wedge P(d) \implies P(Bin\ i\ r\ d)$. Asumimos que vale $P(i) \wedge P(d)$

$P(Bin\ i\ r\ d) = esPreRama\ (Bin\ i\ r\ d)\ xs \Rightarrow length\ xs \leq altura\ (Bin\ i\ r\ d)$

$$\overset{\{E1\}}{\equiv} (\backslash xs \rightarrow null\ xs\ ||\ (r ==\ head\ xs\ \&\&\ (esPreRama\ i\ (tail\ xs)\ ||esPreRama\ d\ (tail\ xs))\ xs \Rightarrow length\ xs \leq altura\ (Bin\ i\ r\ d)$$

De lo cual podemos ver que tenemos 2 casos : el caso en que xs es vacío y el que no

- Caso xs vacía

$$\overset{\{\beta\}}{\equiv} null\ [\,]\ ||\ (r ==\ head\ [\,]\ \&\&\ (esPreRama\ i\ (tail\ [\,])\ ||esPreRama\ d\ (tail\ [\,])) \Rightarrow length\ [\,] \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{\{N0\}}{\equiv} True||\ (r ==\ head\ [\,]\ \&\&\ (esPreRama\ i\ (tail\ [\,])\ ||esPreRama\ d\ (tail\ [\,])) \Rightarrow length\ [\,] \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{\{L0\}}{\equiv} True \Rightarrow 0 \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{\{LEMA\}}{\equiv} True \Rightarrow True$$
$$\equiv True$$

- Caso xs no-vacía (la vamos a representar como $(x : xs)$)

$$\overset{\{\beta\}}{\equiv} null\ (x : xs)\ ||\ (r ==\ head\ (x : xs)\ \&\&\ (esPreRama\ i\ (tail\ (x : xs))\ ||esPreRama\ d\ (tail\ (x : xs))) \Rightarrow length\ (x : xs) \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{\{N1\}}{\equiv} False||\ (r ==\ head\ (x : xs)\ \&\&\ (esPreRama\ i\ (tail\ (x : xs))\ ||esPreRama\ d\ (tail\ (x : xs))) \Rightarrow length\ (x : xs) \leq altura\ (Bin\ i\ r\ d)$$

Que da lugar a dos casos : $r == x$ y $r \neq x$. El segundo no nos interesa ya que hace falso el antecedente y por tanto, da verdadera la fórmula.

* $r == x$

$$\equiv True\ \&\&\ (esPreRama\ i\ (tail\ (x : xs))\ ||esPreRama\ d\ (tail\ (x : xs))) \Rightarrow length\ (x : xs) \leq altura\ (Bin\ i\ r\ d)$$
$$\equiv (esPreRama\ i\ (tail\ (x : xs))\ ||esPreRama\ d\ (tail\ (x : xs))) \Rightarrow length\ (x : xs) \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{HI}{\equiv} True \Rightarrow length\ (x : xs) \leq altura\ (Bin\ i\ r\ d)$$
$$\overset{\{L1,A1\}}{\equiv} True \Rightarrow 1 + length\ xs \leq 1 + \max(altura\ i)(altura\ d)$$
$$\overset{HI}{\equiv} True \Rightarrow 1 \leq 1$$
$$\equiv True$$

Queda probado por inducción sobre AEB.

2. Demostrar con deducción natural sin principios clásicos

$$((P \Rightarrow Q) \land (Q \Rightarrow R)) \Rightarrow \neg R \Rightarrow \neg P$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash (P \implies Q) \land (Q \implies R)}\,\text{ax}}{\Gamma' \vdash P \implies Q}\,\land_{e_1} \quad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash (P \implies Q) \land (Q \implies R)}\,\text{ax}}{\Gamma' \vdash Q \implies R}\,\land_{e_2} \quad \dfrac{}{\Gamma' \vdash \neg R}\,\text{ax}}{\Gamma' \vdash \neg Q}\,\text{MT}}{\Gamma, ((P \Rightarrow Q) \land (Q \Rightarrow R)), \neg R \vdash \neg P}\,\text{MT}}{\Gamma, ((P \Rightarrow Q) \land (Q \Rightarrow R)) \vdash \neg R \Rightarrow \neg P}\,\Rightarrow_i}{\Gamma \vdash ((P \Rightarrow Q) \land (Q \Rightarrow R)) \Rightarrow \neg R \Rightarrow \neg P}\,\Rightarrow_i$$

1.3 Ejercicio 3

Se desea extender el cálculo lambda simplemente tipado para modelar **Diccionarios**. Para eso, se extienden los tipos y expresiones de la siguiente manera:

$$\begin{aligned} \tau &::= \cdots \mid Dicc(\tau, \tau) \\ M &::= \ldots \mid \text{Vacío}_{\sigma,\tau} \mid \text{definir}(M, M, M) \mid \text{def?}(M, M) \mid \text{obtener}(M, M) \end{aligned}$$

- El tipo $\text{Dicc}(o, \tau)$ representa diccionarios con claves de tipo o y valores de tipo τ.
- $\text{Vacío}_{o,\tau}$ es un diccionario vacío con claves de tipo o y valores de tipo τ.
- $\text{definir}(M, N, O)$ define el valor O en el diccionario M para la clave N.
- $\text{def?}(M, N)$ indica si la clave N fue definida en el diccionario M.
- $\text{obtener}(M, N)$ devuelve el valor asociado a la clave N en el diccionario M.(Se espera que el diccionario tenga definida la clave; en caso contrario, la expresión puede tipar, pero no se obtendrá un valor.)

1. Introducir reglas de tipado. Como tenemos nuevos términos, queremos mostrar cuáles son los tipos adecuados. Arranquemos con el más simple :

$$\frac{}{\Gamma \vdash \text{Vacío}_{\sigma,\tau} : Dicc(\sigma, \tau)}\,\text{T-VACIO}$$

Definir lo que hace es básicamente asignar el valor de la tercer coordenada al diccionario que recibe en la primer coordenada en la clave que está en la segunda

$$\frac{\Gamma \vdash M : Dicc(\sigma, \tau) \qquad \Gamma \vdash N : \sigma \qquad \Gamma \vdash O : \tau}{\Gamma \vdash Definir(M, N, O) : Dicc(\sigma, \tau)}\,\text{T-DEFINIR}$$

Para el caso de $def?(M, M)$, este término indica si la clave N fue definida en M (existencia)

$$\frac{\Gamma \vdash M : Dicc(\sigma, \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash def?(M, N) : Bool}\,\text{T-DEF?}$$

El último término $obtener(M, M)$ lo que nos permite es obtener el valor asociado a N en el diccionario M

$$\frac{\Gamma \vdash M : Dicc(\sigma, \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash obtener(M, N) : \tau}\,\text{T-OBTENER}$$

2. Conjunto de Valores - Reglas de cómputo y de Congruencia

- Básicamente agregamos, el Null object (por así decirlo) y el constructor del diccionario
  $V ::= \ldots \mid \text{Vacío}_{\sigma,\tau} \mid definir(V, V, V)$
- Reglas de cómputo
  Veamos **def?** cómo se comporta. Cuando recibe un diccionario vacío y una clave

$$\frac{}{def?(Vacio_{\sigma,\tau}, N) \to False}\ \text{E-DEF?VACIO}$$

Cuando el diccionario no es vacío...

$$\frac{}{def?(definir(M, N, O), L) \to if\ L == N\ then\ True\ else\ False}\ \text{E-DEF?NOVACIO}$$

Para el caso de **obtener**. Tenemos el caso que queremos obtener algo del vacío, lo cual conduce a un estado de error ...

$$\frac{}{obtener(Vacio_{\sigma,\tau}, N) \to ERROR}\ \text{E-OBTENERVACIO}$$

Y tenemos el caso en que la clave puede estar o no

$$\frac{}{obtener(definir(M, N, O), L) \to if\ N == L\ then\ O\ else\ obtener(M, L)}\ \text{E-OBTENER}$$

- Las reglas de congruencia que son para casos de evaluación en los que los términos no son formas normales, todavía, son :
  (a) definir(A,B,C) puede suceder que el A -¿ A' y que sus claves y valores puedan reducirse
  (b) def?(A,B) cuando A -¿ A' por ejemplo si se define un diccionario con cosas que se pueden reducir todavía. El caso B -¿ B' también si como el ejemplo que nos dan a continuación ese Nat se puede evaluar .
  (c) obtener(A,B) es similar a lo que dijimos, A -¿ A' y por supuesto que su clave también.

  Tenemos entonces 7 reglas más para las cuáles definir una congruencia en nuestra extensión.

3. Nos piden reducir :

$(\lambda d : \text{Dicc(Nat, Bool)}.\ \text{if def?}(d, \underline{0})\ \text{then obtener}(d, \underline{0})\ \text{else False})\ \text{definir}(\text{Vacío}_{\text{Nat,Bool}}, \underline{0}, \text{True})$

$\overset{\beta}{\to}$ if def?(definir($\text{Vacío}_{\text{Nat,Bool}}, \underline{0}$, True), $\underline{0}$) then obtener(definir($\text{Vacío}_{\text{Nat,Bool}}, \underline{0}$, True), $\underline{0}$) else False)

$\overset{E-DEF?NOVACIO}{\to}$ if $(if\ \underline{0} == \underline{0}\ then\ True\ else\ False)$ then obtener(definir($\text{Vacío}_{\text{Nat,Bool}}, \underline{0}$, True), $\underline{0}$) else False)

$\to$ if $True$ then obtener(definir($\text{Vacío}_{\text{Nat,Bool}}, \underline{0}$, True), $\underline{0}$) else False)

$\overset{E-IFTRUE}{\to}$ obtener(definir($\text{Vacío}_{\text{Nat,Bool}}, \underline{0}$, True), $\underline{0}$)

$\overset{E-OBTENER}{\to}$ $if\ \underline{0} == \underline{0}\ then\ True\ else\ obtener(Vacio_{\sigma,\tau}, \underline{0})$

$\overset{E-IFTRUE}{\to}$ $True$

2 1C2024 Recu

2.1 Ejercicio 1

En este ejercicio vamos a modelar lógica proposicional en Haskell, de modo de poder construir fórmulas proposicionales y evaluarlas bajo distintas valuaciones.

```
data Prop = Var String | No Prop | Y Prop Prop | O Prop Prop | Imp Prop Prop

type Valuacion = String -> Bool
```

Por ejemplo, la expresión:

```
Y (Var "P") (No (Imp (Var "Q") (Var "R")))
```

representa la proposición lógica: $P \wedge \neg(Q \to R)$. Las valuaciones se representan como funciones que a cada variable proposicional le asignan un valor booleano. Por ejemplo:

```
\x -> x == "P"
```

asigna el valor Verdadero a la variable `P`, y Falso a todas las demás.

1. Nos piden dar el tipo y definir **foldProp** y **recProp** que implementan los esquemas de recursión estructural y primitiva del tipo **Prop**.

```
-- fold (recursi n estructural)
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> (a -> a -> a) -> (a -> a -> a) -> Prop -> a
foldProp fVar fNo fY fO fImp prop = case prop of
        Var s -> fVar s
        No p -> fNo (rec p)
        Y p p' -> fY (rec p) (rec p')
        O p p'-> fO (rec p) (rec p')
        Imp p p'-> fImp (rec p) (rec p')
        where rec = foldProp fVar fNo fY fO fImp

-- la recursi n primitiva lo  nico  que agrega es la posibilidad de mantener referencia sin procesar
     de la estructura
recProp :: (String -> a) -> (Prop -> a -> a) -> (Prop -> Prop -> a -> a -> a) -> (Prop -> Prop -> a
    -> a -> a) -> (Prop -> Prop -> a -> a -> a) -> Prop -> a
recProp fVar fNo fY fO fImp prop = case prop of
        Var s -> fVar s
        No p -> fNo p (rec p)
        Y p p' -> fY p p' (rec p) (rec p')
        O p p'-> fO p p' (rec p) (rec p')
        Imp p p'-> fImp p p' (rec p) (rec p')
        where rec = recProp fVar fNo fY fO fImp
```

2. Para el punto b nos piden definir la función **variables** que básicamente devuelve, sin repetidos, las variables de cierta proposición

```
variables :: Prop -> [String]
variables prop = eliminarRepetidos (foldProp (\s -> [s]) (\p -> p) (++) (++) (++) prop)
-- ejemplo :
ej1 = (O (Var "P") (No(Y (Var "Q") (Var "P"))))

eliminarRepetidos :: Eq a => [a] -> [a]
eliminarRepetidos [] = []
eliminarRepetidos (x:xs) = if (elem x xs) then eliminarRepetidos xs else x : eliminarRepetidos xs
```

3. La función **evaluar** indica el valor de verdad de la fórmula

```
-- evaluar
evaluar :: Valuacion -> Prop -> Bool
evaluar val = foldProp val (not) (&&) (||) (\p q -> not p || q)
```

4. La función **estaEnFNN** nos indica si la fórmula está en su forma normal negada (sin implicaciones y cuando la negación sólo seaplica a variables) *(consultar)*

2.2 Ejercicio 2

Considerar las siguientes definiciones sobre árboles con información en las hojas

```
data AIH a = Hoja a | Bin (AIH a) (AIH a)

    der :: AIH a -> AIH a
{D} der (Bin _ d) = d

    esHoja :: AIH a -> Bool
{E0}esHoja (Hoja _) = True
{E1}esHoja (Bin _ _) = False

    mismaEstructura :: AIH a -> (AIH a -> Bool)
{M0}mismaEstructura (Hoja _) = esHoja
{M1}mismaEstructura (Bin i d) = \t ->
      not (esHoja t) &&
      mismaEstructura i (izq t) &&
      mismaEstructura d (der t)

    izq :: AIH a -> AIH a
{I} izq (Bin i _) = i
```

Nos piden demostrar:

$$\forall t, u : AIH\ a.\quad mismaEstructura\ t\ u = mismaEstructura\ u\ t$$

1. Primer paso de la demostración es decidir qué vamos a hacer. En este caso, vamos a hacer inducción sobre **AIH a**. En particular, sobre el primero. Entonces, decimos

$$\forall x, u : AIH\ a.\quad mismaEstructura\ x\ u = mismaEstructura\ u\ x$$

- **QVQ** caso base :

$$\begin{aligned} & mismaEstructura\ (Hoja\ a)\ u = mismaEstructura\ u\ (Hoja\ a) \\ \overset{\{M0\}}{\equiv}\ & esHoja\ u = mismaEstructura\ u\ (Hoja\ a) \end{aligned}$$

  Esto nos lleva a destrabar a **u** por extensionalidad y separar en 2 casos :

  - Caso **u es Hoja**

$$\begin{aligned} & esHoja\ (Hoja\ a) = mismaEstructura\ (Hoja\ a)\ (Hoja\ a) \\ \overset{\{E0\}}{\equiv}\ & True = mismaEstructura\ (Hoja\ a)\ (Hoja\ a) \\ \overset{\{M0\}}{\equiv}\ & True = esHoja\ (Hoja\ a) \\ \overset{\{E0\}}{\equiv}\ & True = True \\ \equiv\ & True \end{aligned}$$

  - Caso **u no es Hoja**

$$\begin{aligned} & esHoja\ (Bin\ (AIH\ a)\ (AIH\ a)) = mismaEstructura\ (Bin\ (AIH\ a)\ (AIH\ a))\ (Hoja\ a) \\ \overset{\{E1\}}{\equiv}\ & False = mismaEstructura\ (Bin\ (AIH\ a)\ (AIH\ a))\ (Hoja\ a) \\ \overset{\{M1\}}{\equiv}\ & False = (\backslash t- > not\ (esHoja\ t)\ \&\&\ mismaEstructura\ i\ (izq\ t)\&\&\ mismaEstructura\ d\ (der\ t))\ (Hoja\ a) \\ \overset{\beta}{\equiv}\ & False = (not\ (esHoja\ (Hoja\ a))\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a) \\ \overset{\{E0\}}{\equiv}\ & False = (not\ (True)\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a))) \\ \overset{not}{\equiv}\ & False = False\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a))) \\ \equiv\ & False = False \\ \equiv\ & True \end{aligned}$$

- Ahora que probamos para el caso base, **QVQ**

$$\forall i, d : AIH\ a, r : a.\quad P(i) \wedge P(d) \implies P(Bin\ i\ r\ d)$$

  Con esto, queremos demostrar que :

$$P(Bin\ i\ r\ d) = mismaEstructura\ (Bin\ i\ r\ d)\ u = mismaEstructura\ u\ (Bin\ i\ r\ d)$$

  Asumimos que valen $P(i) \wedge P(d)$ y serán nuestras HI :

$$\begin{aligned} P(i) &= mismaEstructura\ i\ u = mismaEstructura\ u\ i \\ P(d) &= mismaEstructura\ d\ u = mismaEstructura\ u\ d \end{aligned}$$

  Con esto en mente, procedemos a demostrarlo:

$$\begin{aligned} & mismaEstructura\ (Bin\ i\ r\ d)\ u = mismaEstructura\ u\ (Bin\ i\ r\ d) \\ \overset{\{M1\}}{\equiv}\ & (\backslash t- > not\ (esHoja\ t)\ \&\&\ mismaEstructura\ i\ (izq\ t)\&\&\ mismaEstructura\ d\ (der\ t))\ u = \ldots \\ \overset{\beta}{\equiv}\ & not\ (esHoja\ u)\ \&\&\ mismaEstructura\ i\ (izq\ u)\&\&\ mismaEstructura\ d\ (der\ u) = \ldots \end{aligned}$$

  Esto nos traba la situación y por extensionalidad separamos en dos casos :

– Caso **u es Hoja**

$not\ (esHoja\ (Hoja\ a))\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a)) = \ldots$

$\overset{\{E0\}}{\equiv} not\ (True)\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a)) = \ldots$

$\overset{not}{\equiv} False\ \&\&\ mismaEstructura\ i\ (izq\ (Hoja\ a))\&\&\ mismaEstructura\ d\ (der\ (Hoja\ a)) = \ldots$

$\equiv False = mismaEstructura\ (Hoja\ a)\ (Bin\ i\ r\ d)$

$\overset{\{M0\}}{\equiv} False = esHoja\ (Bin\ i\ r\ d)$

$\overset{\{E1\}}{\equiv} False = False$

$\equiv True$

– Caso **u no es Hoja**

$not\ (esHoja\ (Bin\ (AIH\ a)\ (AIH\ a)))\ \&\&\ mismaEstructura\ i\ (izq\ (Bin\ (AIH\ a)\ (AIH\ a)))$
$\&\&mismaEstructura\ d\ (der\ (Bin\ (AIH\ a)\ (AIH\ a))) = \ldots$

$\overset{\{E1\},not}{\equiv} True\&\&\ mismaEstructura\ i\ (izq\ (Bin\ (AIH\ a)\ (AIH\ a)))\&\&$
$mismaEstructura\ d\ (der\ (Bin\ (AIH\ a)\ (AIH\ a))) = \ldots$

$\equiv mismaEstructura\ i\ (izq\ (Bin\ (AIH\ a)\ (AIH\ a)))\&\&mismaEstructura\ d\ (der\ (Bin\ (AIH\ a)\ (AIH\ a))) = \ldots$

$\overset{\{I\},\{D\}}{\equiv} mismaEstructura\ i\ (iu)\&\&mismaEstructura\ d\ (du) = \ldots \quad$ `{donde iu, du : AIH a}`

$\overset{HI}{\equiv} True\&\&True = \ldots$

$\equiv True = mismaEstructura\ (Bin\ (AIH\ a)\ (AIH\ a))\ (Bin\ i\ r\ d)$

$\overset{\{M1\}}{\equiv} True = (\backslash t- > not\ (esHoja\ t)\ \&\&\ mismaEstructura\ iu\ (izq\ t)\&\&\ mismaEstructura\ du\ (der\ t))\ (Bin\ i\ r\ d)$

$\overset{\beta}{\equiv} True = not(esHoja\ (Bin\ i\ r\ d))\&\&\ mismaEstructura\ iu(izq(Bin\ i\ r\ d))\&\&mismaEstructura\ du(der(Bin\ i\ r\ d$

$\overset{\{E1\},not}{\equiv} True = True\&\&\ mismaEstructura\ iu\ (izq\ (Bin\ i\ r\ d))\&\&\ mismaEstructura\ du\ (der\ (Bin\ i\ r\ d))$

$\overset{\{I\},\{D\}}{\equiv} True = mismaEstructura\ iu\ i\ \&\&\ mismaEstructura\ du\ d$

$\overset{HI}{\equiv} True = mismaEstructura\ True\ \&\&\ True$

$\equiv True = True$

$\equiv True$

2.3 Ejercicio 3

Se desea extender el cálculo lambda simplemente tipado para modelar **Árboles con info en las hojas**. Para eso se extienden tipos y expresiones como sigue :

$$\tau ::= \ldots \mid AIH(\tau)$$
$$M ::= \ldots \mid Hoja(M) \mid Bin(M, M) \mid case\ M\ of\ Hoja\ x \rightsquigarrow M; Bin(i, d) \rightsquigarrow M$$

a) Nos piden introducir las reglas de tipado : tenemos que mirar los nuevos términos

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash Hoja(M) : AIH(\tau)}\ \text{T-HOJA}$$

$$\frac{\Gamma \vdash M_1 : AIH(\tau) \qquad \Gamma \vdash M_2 : AIH(\tau)}{\Gamma \vdash Bin(M_1, M_2) : AIH(\tau)}\ \text{T-BIN}$$

$$\frac{\Gamma \vdash\ M_1 : AIH(\tau) \qquad \Gamma, x : \tau \vdash\ M_2 : \sigma \qquad \Gamma, i : AIH(\tau), d : AIH(\tau) \vdash\ M_3 : \sigma}{\Gamma \vdash case\ M_1\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i, d) \rightsquigarrow M_3 : AIH(\tau)}\ \text{T-CASE}$$

b) El nuevo conjunto de valores, las reglas de cómputo y congruencia

- Conjunto de valores

$$V ::= \ldots \mid Hoja(V) \mid Bin(V, V)$$

- Reglas de cómputo Para **case .. of ...** tenemos el caso en que M1 es Hoja (y el subtérmino es valor (estamos en reglas de cómputo))

$$\frac{}{case\ (Hoja(V))\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to M2\{x := V\}}\ \text{E-CASEHOJA1}$$

Y el caso en que M1 es un Bin(V,V)

$$\frac{}{case\ (Bin(V_1, V_2))\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to M3\{i := V_1\}\{d := V_2\}}\ \text{E-CASEBIN1}$$

Ahora bien, hay escenarios en los que los subtérminos de los extendidos a nuestra gramática no están en su forma normal y tendremos que evaluarlos, para ello se definen ...

- Reglas de congruencia

$$\frac{M \to M'}{case\ M\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to case\ M'\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3}\ \text{E-CASE}$$

$$\frac{M \to M'}{case\ (Hoja(M))\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to M2\{x := M'\}}\ \text{E-CASEHOJA2}$$

$$\frac{N \to N'}{case\ (Bin(N, V_2))\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to M3\{i := N'\}\{d := V_2\}}\ \text{E-CASEBIN2}$$

$$\frac{N \to N'}{case\ (Bin(V_1, N))\ of\ Hoja\ x \rightsquigarrow M_2; Bin(i,d) \rightsquigarrow M_3 \to M3\{i := V_1\}\{d := N'\}}\ \text{E-CASEBIN3}$$

c) Cómo se reduce :

$$\begin{aligned}
&case\ (\lambda n : Nat.Hoja(n))\ Succ(zero)\ of\ Hoja\ x \rightsquigarrow Succ(Pred(x)); Bin(i,d) \rightsquigarrow zero \\
&\overset{E-CASE}{\to} case\ Hoja(Succ(zero))\ of\ Hoja\ x \rightsquigarrow Succ(Pred(x)); Bin(i,d) \rightsquigarrow zero \\
&\overset{E-CASEHOJA1}{\to} Succ(Pred(x))\{x := Succ(zero)\} \\
&\to Succ(Pred(Succ(zero))) \\
&\overset{E-PREDSUCC}{\to} Succ(zero))
\end{aligned}$$

3 2C2024 Parcial

3.1 Ejercicio 1

El siguiente tipo de datos sirve para representar un *buffer con historia* que permite escribir o leer en cualquiera de sus posiciones (las posiciones tienen tipo `Int`). El tipo del buffer es paramétrico en el tipo de los contenidos que se pueden guardar en él. Si se escribe dos veces en la misma posición, el nuevo contenido pisa al anterior (por simplicidad). La lectura elimina el contenido leído.

```
data Buffer a = Empty | Write Int a (Buffer a) | Read Int (Buffer a)
```

Definimos el siguiente buffer para los ejemplos:

```
buf = Write 1 'a' $ Write 2 'b' $ Write 1 'c' $ Empty
```

a) Nos piden definir **foldBuffer** y **recBuffer**

```
foldBuffer :: b -> (Int-> a -> b -> b) -> (Int-> b -> b) -> Buffer a -> b
foldBuffer fEmpty fWrite fRead buf = case buf of
    Empty -> fEmpty
    Write n s b -> fWrite n s (rec b)
    Read n b -> fRead n (rec b)
    where rec = foldBuffer fEmpty fWrite fRead

recBuffer :: b -> (Int-> a -> Buffer a -> b -> b) -> (Int-> Buffer a -> b -> b) -> Buffer a -> b
recBuffer fEmpty fWrite fRead buf = case buf of
    Empty -> fEmpty
    Write n s b -> fWrite n s b (rec b)
    Read n b -> fRead n b (rec b)
    where rec = recBuffer fEmpty fWrite fRead
```

b) Nos piden definir una función que devuelva una lista con las **posiciones ocupadas** del buffer sin repetir

```
posicionesOcupadas :: Buffer a -> [Int]
posicionesOcupadas = foldBuffer [] (\n s rec -> if (elem n rec) then rec else rec ++ [n])
                     (\n rec -> rec)
```

c) Necesitamos definir la función que devuelve el **contenido** de una posición del buffer

```
contenido :: Int -> Buffer a -> Maybe  a
contenido n = recBuffer Nothing (\m s l rec -> if n == m then Just s else Nothing)
            (\m l rec -> Nothing)
```

d) La función **completarLecturas** indica si se puede leer exitosamente en todas las lecturas del buffer

```
puedeCompletarLecturas :: (Eq a) => Buffer a -> Bool
puedeCompletarLecturas = recBuffer False (\m s l rec -> rec )
                         (\m l rec -> if (contenido m l) /= Nothing then True else False )
```

e) La función **deshacer** nos permite deshacer las últimas n-operaciones en el buffer

```
deshacer :: Buffer a -> Int -> Buffer a
deshacer = recBuffer (const Empty) (\m s l rec -> \n-> if n == 0 then Write m s l else rec (n-1)) (\m
      l rec -> \n -> if n == 0 then Read m l else rec (n-1))

deshacerCheck = contenido 1 (deshacer buf 2) ~> Just 'c'
```

La idea de usar currificación es para que podemos usar el número n para iterar. Usamos recBuffer para tener referencia a toda la estructura

3.2 Ejercicio 2

Considerar las siguientes definiciones

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

    const :: a -> b -> a
{C} const = (\x -> \y -> x )

    altura :: AB a -> Int
{A0}altura Nil = 0
{A1}altura (Bin i r d) = 1 + max (altura i) (altura d)
    zipAB :: AB a -> AB b -> AB (a,b)
{Z0}zipAB Nil = const Nil
{Z1}zipAB (Bin i r d) = (\t -> case t of Nil -> Nil Bin i' r' d' -> Bin (zipAB i i') (r r') (zipAB d d'))
```

a) Nos piden demostrar

$$\forall t, u : AB\ a. \quad altura\ t \geq altura\ (zipAB\ t\ u)$$

Contamos con este **lema** ya demostrado

$$\{LEMA\}\ \forall t : AB\ a.\ altura\ t \geq 0$$

Vamos a demostrarlo por inducción sobre **AB a**. Enunciamos

$$\forall x : AB. \quad P(x) = altura\ x \geq altura\ (zipAB\ x\ u)$$

- Caso base :

$$\begin{aligned} P(Nil) &= altura\ Nil \geq altura\ (zipAB\ Nil\ u) \\ &\overset{\{A0\}}{\equiv} 0 \geq altura\ (zipAB\ Nil\ u) \\ &\overset{\{Z0\}}{\equiv} 0 \geq altura\ Const\ Nil \\ &\overset{\{C\}}{\equiv} 0 \geq altura\ Nil \\ &\overset{\{A0\}}{\equiv} 0 \geq 0 \\ &\equiv True \end{aligned}$$

– Ahora podemos enunciar $\forall i, d : AB\ a, r : a.\quad P(i) \wedge P(d) \implies P(Bin\ i\ r\ d)$. Asumimos verdaderas $P(i) \wedge P(d)$ y serán nuestras **HI**. Con esto **QVQ**

$$
\begin{aligned}
P(Bin\ i\ r\ d) &= altura\ (Bin\ i\ r\ d) \geq altura\ (zipAB\ (Bin\ i\ r\ d)\ u) \\
&\overset{\{A1\}}{\equiv} 1 + \max(altura\ i)(altura\ d) \geq altura\ (zipAB\ (Bin\ i\ r\ d)\ u) \\
&\overset{\{Z1\}}{\equiv} \ldots \geq altura\ (\backslash t\ case\ t\ of Nil \to Nil\ \ Bin\ i'\ r'\ d' \to (zipAB\ i\ i')\ (r\ r')\ (zipAB\ d\ d'))\ u) \\
&\overset{\beta}{\equiv} \ldots \geq altura\ (case\ u\ of Nil \to Nil\ \ Bin\ ui\ ur\ ud \to (zipAB\ i\ ui)\ (r\ ur)\ (zipAB\ d\ ud))
\end{aligned}
$$

De esto se desprenden dos casos, que los resolvemos por extensionalidad

* Caso **u es Nil**

$$
\begin{aligned}
&\ldots \geq altura\ (case\ Nil\ of Nil \to Nil\ \ Bin\ ui\ ur\ ud \to (zipAB\ i\ ui)\ (r\ ur)\ (zipAB\ d\ ud)) \\
&\overset{case\_Nil}{\equiv} \ldots \geq altura\ Nil \\
&\overset{\{A0\}}{\equiv} \ldots \geq 0 \\
\equiv\ & 1 + \max(altura\ i)(altura\ d) \geq 0 \\
&\overset{\{LEMA\}}{\equiv} True
\end{aligned}
$$

* Caso **u no es Nil**

$$
\begin{aligned}
&\ldots \geq altura\ (case\ Bin\ ui\ ur\ ud\ of Nil \to Nil\ \ Bin\ ui\ ur\ ud \to (zipAB\ i\ ui)\ (r\ ur)\ (z \\
&\overset{case\_Bin}{\equiv} \ldots \geq altura((zipAB\ i\ ui)\ (r\ ur)\ (zipAB\ d\ ud)) \\
\equiv\ & 1 + \max(altura\ i)(altura\ d) \geq altura((zipAB\ i\ ui)\ (r\ ur)\ (zipAB\ d\ ud)) \\
\overset{\{A1\}}{\equiv}\ & 1 + \max(altura\ i)(altura\ d) \geq 1 + \max(altura(zipAB\ i\ ui))\ (altura(zipAB\ d\ ud))
\end{aligned}
$$

Recordemos las HI :

$$
\begin{aligned}
P(i) &= altura\ i \geq altura(zipAB\ i\ u) \\
P(d) &= altura\ d \geq altura(zipAB\ d\ u)
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\{HI\}}{\equiv} 1 \geq 1 \\
&\equiv True
\end{aligned}
$$

b) Demostrar sin principios clásicos :

$$((\rho \wedge \sigma) \vee (\rho \wedge \tau)) \implies (\sigma \wedge \rho) \vee \tau$$

$$
\dfrac{\dfrac{}{\Gamma' \vdash (\rho \wedge \sigma) \vee (\rho \wedge \tau)}\,\text{ax} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma', (\rho \wedge \sigma) \vdash \rho \wedge \sigma}\,\text{ax}}{\Gamma', (\rho \wedge \sigma) \vdash \sigma}\,\wedge_{e_2} \qquad \dfrac{\dfrac{}{\Gamma', (\rho \wedge \sigma) \vdash \rho \wedge \sigma}\,\text{ax}}{\Gamma', (\rho \wedge \sigma) \vdash \rho}\,\wedge_{e_1}}{\Gamma', (\rho \wedge \sigma) \vdash \sigma \wedge \rho}\,\wedge_i}{\Gamma', (\rho \wedge \sigma) \vdash (\sigma \wedge \rho) \vee \tau}\,\vee_{i_1} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma', (\rho \wedge \tau) \vdash \rho \wedge \tau}\,\text{ax}}{\Gamma', (\rho \wedge \tau) \vdash \tau}\,\wedge_{e_2}}{\Gamma', (\rho \wedge \tau) \vdash (\sigma \wedge \rho) \vee \tau}\,\vee_{i_2}}{\dfrac{\Gamma, ((\rho \wedge \sigma) \vee (\rho \wedge \tau)) \vdash (\sigma \wedge \rho) \vee \tau}{\Gamma \vdash ((\rho \wedge \sigma) \vee (\rho \wedge \tau)) \implies (\sigma \wedge \rho) \vee \tau}\,\Rightarrow_i}\,\vee_e
$$

3.3 Ejercicio 3

Se desea extender el cálculo lambda simplemente tipado para modelar **Árboles ternarios**. Por eso, se extienden los tipos y expresiones como sigue:

$$
\begin{aligned}
\tau &::= \ldots \mid AT(\tau) \\
M &::= \ldots \mid TNil_\tau \mid Tern(M, M, M, M) \mid foldAT\ M\ TNil \rightsquigarrow M; Tern(x, ri, rm, rd) \rightsquigarrow M
\end{aligned}
$$

a) Nos piden introducir las reglas de tipo para esta extensión

$$\frac{}{\Gamma \vdash TNil_\tau : AT(\tau)}\ \text{T-TNIL}$$

$$\frac{\Gamma \vdash A : \tau \qquad \Gamma \vdash B : AT(\tau) \qquad \Gamma \vdash C : AT(\tau) \qquad \Gamma \vdash D : AT(\tau)}{\Gamma \vdash Tern(A, B, C, D) : AT(\tau)}\ \text{T-TERN}$$

$$\frac{\Gamma \vdash A : AT(\tau) \qquad \Gamma \vdash B : \rho \qquad \Gamma, x : \tau, ri : \rho, rm : \rho, rd : \rho \vdash C : \rho}{\Gamma \vdash foldAT\ A\ TNil \rightsquigarrow B; Tern(x, ri, rm, rd) \rightsquigarrow C : \rho}\ \text{T-FOLD}$$

b) Nos piden dar el conjunto de valores extendido

$$V ::= \ldots \mid TNil_\tau \mid Tern(V, V, V, V)$$

las reglas de cómputo...

$$\frac{}{foldAT\ TNil_\tau\ TNil \rightsquigarrow B; Tern(x, ri, rm, rd) \rightsquigarrow C \to B}\ \text{E-FOLDNIL}$$

$$\frac{}{foldAT\ Tern(V_1, V_2, V_3, V_4)\ TNil \rightsquigarrow B; Tern(x, ri, rm, rd) \rightsquigarrow C}\ \text{E-FOLDVALUES}$$

$\to C\{x := V_1\}\{ri := foldAT\ V_2 \ldots\}\{rm := foldAT\ V_3 \ldots\}\{rd := foldAT\ V_4 \ldots\}$ Las de congruencias son todas aquellas en las que $M \to M'$ o sea :
Para el Tern...

- $Tern(M, V_2, V_3, V_4) \to Tern(M', V_2, V_3, V_4)$
- $Tern(O, M, V_3, V_4) \to Tern(O, M', V_3, V_4)$
- $Tern(O, P, M, V_4) \to Tern(O, P, M', V_4)$
- $Tern(O, P, Q, M) \to Tern(O, P, Q, M')$

Para el foldAT...

- $foldAT\ M\ TNil \rightsquigarrow B; Tern(x, ri, rm, rd) \rightsquigarrow C \to foldAT\ M'\ TNil \rightsquigarrow B; Tern(x, ri, rm, rd) \rightsquigarrow C$

- Reducir :

$(\lambda t : AT(Nat).\ foldAT\ TNil \rightsquigarrow False;\ Tern(x, ri, rm, rd) \rightsquigarrow isZero(Pred(x)))\ \ Tern(\underline{1}, TNil_{Nat}, TNil_{Nat}, TNil_{Nat})$
$\overset{\beta}{\to} foldAT\ Tern(\underline{1}, TNil_{Nat}, TNil_{Nat}, TNil_{Nat})\ TNil \rightsquigarrow False;\ Tern(x, ri, rm, rd) \rightsquigarrow isZero(Pred(x)))$
$\overset{E-FOLDVALUES}{\to} isZero(Pred(\underline{1}))$
$\equiv isZero(Pred(Succ(zero)))$
$\overset{E-PREDSUCC}{\to} isZero(zero)$
$\overset{E-ISZEROZERO}{\to} True$

4 2C2024 Recu

4.1 Ejercicio 1

En este ejercicio no se permite utilizar recursión explícita, a menos que se indique lo contrario. El siguiente tipo de datos sirve para representar operadores que realizan operaciones combinadas sobre números enteros. Por simplicidad, modelaremos solamente las sumas y divisiones enteras:

```
data Operador = Sumar Int | DividirPor Int | Secuencia [Operador]
```

`Sumar n` representa la operación que suma `n` a un número entero. `DividirPor n` representa la operación que divide a un entero por `n` (descartando el resto). `Secuencia ops` representa la composición a izquierda de todas las operaciones en `ops`. En otras palabras, representa la operación de aplicar a un número todas las operaciones en `ops` de izquierda a derecha, siendo resultado de cada operación la entrada de la siguiente. Por ejemplo:

```
Secuencia [Sumar 5, DividirPor 2]
```

representa la operación que, dado un entero, le suma 5, y al resultado lo divide por 2.

a) Nos piden foldear la estructura con **foldOperador**

```haskell
-- foldOperador
foldOperador :: (Int -> a) -> (Int -> a) -> ([a] -> a) -> Operador -> a
foldOperador fSuma fDividir fSecuencia operator = case operator of
    Sumar n -> fSuma n
    DividirPor n -> fDividir n
    Secuencia op -> fSecuencia (map rec op)
        where rec = foldOperador fSuma fDividir fSecuencia
```

b) Tenemos que indicar si **falla** cuando hay una división por cero

```haskell
falla :: Operador -> Bool
falla = foldOperador (\n -> False) (\i -> if i == 0 then True else False) (\l -> or l )
```

c) Queremos **aplanar** las subsecuencias para sólo dejar una : Nos recomiendan usar *concatMap* y hacer una auxiliar para usar con ella.

```haskell
aplanar :: Operador -> Operador
aplanar = foldOperador (\o -> Sumar o) (\o -> DividirPor o) (\l -> Secuencia (concatMap listar l))
    where
        listar (Secuencia lista) = concatMap listar lista   -- buscamos aplanar recursivamente
        listar operacion = [operacion]                      -- caso base cuando ya no hay secuencia
```

Básicamente lo que hacemos es, en el caso peludo de la secuencias, agarrar a cada "elemento simple" (sumar / dividir) y lo convertimos en una lista de un elemento . Después el concatMap se encarga de aplicarle esa auxiliar que aplana a cada subsecuencia y unificar todo en una sola lista.

d) No sé si entendí bien el ejemplo, pero la idea es componer de izquierda a derecha y por eso pensé en *foldl*

```haskell
componerTodas ::  [a -> a] -> (a -> a)
componerTodas = foldl (.) id
```

Caso lista vacía devuelve *id*, sino, va componiendo de izquierda a derecha

e) *incompleto*

4.2 Ejercicio 2

Considerar las siguientes definiciones:

```haskell
    const :: a -> b -> a
{C} const = (\x -> \y -> x)

    head :: [a] -> a
{H} head (x:xs) = x

    tail :: [a] -> [a]
{T} tail (x:xs) = xs

    length :: [a] -> Int
{L0}length [] = 0
{L1}length (x:xs) = 1 + length xs

    null :: [a] -> Bool
{N0}null [] = True
{N1}null (x:xs) = False

    zip :: [a] -> [b] -> [(a,b)]
{Z0}zip [] = const []
{Z1}zip (x:xs) = \ys -> if null ys then [] else (x, head ys) : zip xs (tail ys)
```

a) Nos piden demostrar

$$\forall xs, ys : [a]. \quad length\ (zip\ xs\ ys) = \min(length\ xs)(length\ ys)$$

Vamos a proceder con inducción sobre el tipo **[a]**. Para ello, enunciamos

$\forall x : a, xs, ys : [a]. \quad P(xs) = length\ (zip\ xs\ ys) = \min(length\ xs)(length\ ys)$. Queremos ver que $P(xs) \implies P(x : xs)$

– Caso Base : $P([\,])$

$$\begin{aligned} P([\,]) &= length\ (zip\ [\,]\ ys) = \min(length\ [\,])(length\ ys) \\ &\overset{\{L0\}}{\equiv} length\ (zip\ [\,]\ ys) = \min(0)(length\ ys) \\ &\overset{\{Z0\}}{\equiv} length\ (const\ [\,]) = \min(0)(length\ ys) \\ &\overset{\{C\}}{\equiv} length\ [\,] = \min(0)(length\ ys) \\ &\overset{\{L0\}}{\equiv} 0 = \min(0)(length\ ys) \end{aligned}$$

Que da lugar a dos casos, cuando

1. **ys es vacío**

$$\begin{aligned} 0 &= \min(0)(length\ [\,]) \\ &\overset{\{L0\}}{\equiv} 0 = \min(0)(0) \\ &\overset{min}{\equiv} 0 = 0 \\ &\equiv True \end{aligned}$$

2. **ys es no vacío**

$$\begin{aligned} 0 &= \min(0)(length\ ys) \text{ en particular length ys } > 0 \\ &\overset{min}{\equiv} 0 = 0 \\ &\equiv True \end{aligned}$$

– Paso inductivo. Como dijimos, queremos probar que $P(xs) \implies P(x : xs)$. Asumimos verdadera $P(xs)$ y por tanto, será nuestra **HI**:

$$P(xs) = length\ (zip\ xs\ ys) = \min(length\ xs)(length\ ys)$$

$$\begin{aligned} P(x : xs) &= length\ (zip\ (x : xs)\ ys) = \min(length\ (x : xs))(length\ ys) \\ &\overset{\{L1\}}{\equiv} length\ (zip\ (x : xs)\ ys) = \min(1 + length\ xs))(length\ ys) \\ &\overset{\{Z1\}}{\equiv} length\ ((\backslash l \to if\ null\ l\ then\ [\,]\ else\ (x,\ head\ l) : zip\ xs\ (tail\ l))\ ys) = \ldots \\ &\overset{\beta}{\equiv} length\ (if\ null\ ys\ then\ [\,]\ else\ (x,\ head\ ys) : zip\ xs\ (tail\ ys)) = \ldots \end{aligned}$$

Lo que nos lleva dos casos,

1. Caso **ys vacía**

$$\begin{aligned} &length\ (if\ null\ [\,]\ then\ [\,]\ else\ (x,\ head\ [\,]) : zip\ xs\ (tail\ [\,])) = \min(1 + length\ xs))(length\ [\,]) \\ &\overset{\{N1\}}{\equiv} length([\,]) = \min(1 + length\ xs))(length\ [\,]) \\ &\overset{\{L0\}}{\equiv} 0 = \min(1 + length\ xs))(0) \\ &\overset{\{LEMA\}}{\equiv} 0 = 0 \\ &\equiv True \end{aligned}$$

2. Caso **ys no vacía**

$$\begin{aligned} &length\ (if\ null\ (y : ys)\ then\ [\,]\ else\ (x,\ head\ (y : ys)) : zip\ xs\ (tail\ (y : ys))) = \ldots \\ &\equiv length\ ((x,\ head\ (y : ys)) : zip\ xs\ (tail\ (y : ys))) = \ldots \\ &\overset{\{H\}\{T\}}{\equiv} length\ ((x,\ y) : zip\ xs\ ys) = \ldots \\ &\overset{\{L1\}}{\equiv} 1 + length\ (zip\ xs\ ys) = \min(1 + length\ xs)(length\ (y : ys)) \\ &\overset{\{L1\}}{\equiv} 1 + length\ (zip\ xs\ ys) = \min(1 + length\ xs)(1 + length\ ys) \\ &\overset{\{HI\}}{\equiv} True \text{ ya que el 1 se suma en todos los subtérminos de un lado y del otro de la igualdad} \end{aligned}$$

b) Nos piden demostrar el siguiente teorema. Vale usar principios clásicos

$$(\tau \implies (\sigma \wedge \rho)) \vee (\rho \implies (\sigma \implies \tau))$$

EL truco para este ejercicio es que sabemos que :

- $Falso \implies \star \; es \; Verdadero$
- $\star \implies \star \; es \; Verdadero$

Con esto en mente, elegimos que $\star = \tau$ y entonces ...

$$\dfrac{\dfrac{}{\vdash \tau \vee \neg\tau}\,\text{LEM} \qquad \dfrac{\dfrac{\dfrac{}{\tau, \rho, \sigma \vdash \tau}\,\text{ax}}{\tau, \rho \vdash \sigma \implies \tau}\Rightarrow_i}{\tau \vdash (\tau \implies (\sigma \wedge \rho)) \vee (\rho \implies (\sigma \implies \tau))}\vee_{i_2} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg\tau, \tau \vdash \tau}\,ax \quad \dfrac{}{\neg\tau, \tau \vdash \neg\tau}\,ax}{\neg\tau, \tau \vdash \bot}\neg_e}{\neg\tau, \tau \vdash \sigma \wedge \rho}\bot_e}{\neg\tau \vdash \tau \implies (\sigma \wedge \rho)}\Rightarrow_i}{\neg\tau \vdash (\tau \implies (\sigma \wedge \rho)) \vee (\rho \implies (\sigma \implies \tau))}\vee_{i_1}}{\vdash (\tau \implies (\sigma \wedge \rho)) \vee (\rho \implies (\sigma \implies \tau))}$$

4.3 Ejercicio 3

Se extenderán los tipos y términos de la siguiente manera:

$$\tau ::= \cdots \mid \text{Cola } \tau$$
$$M ::= \cdots \mid \langle\rangle_\tau \mid M \bullet M \mid \text{recr } M \triangleright \langle\rangle \rightsquigarrow M; r, c \bullet x \rightsquigarrow M$$

a) Introducimos las reglas de tipado

$$\frac{}{\Gamma \vdash \langle\rangle_\tau : Cola_\tau}\,\text{T-COLAEMPTY}$$

$$\frac{\Gamma \vdash M_1 : Cola_\tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 \bullet M_2 : Cola_\tau}\,\text{T-ENCOLAR}$$

$$\frac{\Gamma \vdash M_1 : Cola_\tau \qquad \Gamma \vdash M_2 : \sigma \qquad \Gamma, r : \sigma, c : Cola_\tau, x : \tau \vdash M_3 : \sigma}{\Gamma \vdash \text{recr } M_1 \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3 : \sigma}\,\text{T-RECR}$$

b) demostrar validez del siguiente juicio

$$\dfrac{\dfrac{\dfrac{}{c : Cola_{Nat} \vdash c : Cola_{Nat}}\,\text{ax} \quad \dfrac{}{c : Cola_{Nat} \vdash \langle\rangle_{Bool} : Cola_{Bool}}\,\text{T-C.E} \quad \dfrac{\dfrac{}{\Gamma' \vdash r : Cola_{Bool}}\,\text{ax} \quad \dfrac{}{\Gamma' \vdash True : Bool}\,\text{ax}}{c : Cola_{Nat}, r : Cola_{Bool}, y : Cola_{Nat}, x : Nat \vdash r \bullet True : Cola_{Bool}}\,\text{T-ENCOLAR}}{c : Cola_{Nat} \vdash \text{recr } c \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{Bool}; r, y \bullet x \rightsquigarrow r \bullet True : Cola_{Bool}}\,\text{T-RECR}}{\vdash \lambda c : Cola_{Nat}.\text{recr } c \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{Bool}; r, y \bullet x \rightsquigarrow r \bullet True : (Cola_{Nat} \to Cola_{Bool})}\,\text{T-ABS}$$

c) Conjunto de valores y reglas de cómputo

$$V ::= \ldots \mid \langle\rangle_\tau \mid V \bullet V$$

Reglas de cómputo

$recr \; \langle\rangle_\tau \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3 \to M_2$ `E-RECREMPTY`

$recr \; V_1 \bullet V_2 \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3 \to M_3\{r := recr \; V_1 \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3\}\{c := V_1\}\{x := V_2\}$ `E-RECR1`

Reglas de congruencia **M → M'**

$M \bullet V \to M' \bullet V$ `E-ENCOLAR1`

$V \bullet M \to V \bullet M'$ `E-ENCOLAR2`

$recr \; M \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3 \to recr \; M' \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3$ `E-RECR2`

- Reducir :

$$
\begin{aligned}
&recr\ \langle\rangle_{Nat}\ \bullet zero \bullet \underline{1} \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{Nat};\ r, c \bullet x \rightsquigarrow if\ isZero(x)\ then\ c\ else\ r \bullet x \\
&\overset{E-RECR1}{\rightarrow} if\ isZero(\underline{1})\ then\ (\langle\rangle_{Nat}\ \bullet zero)\ else\ (recr\ \langle\rangle_{Nat}\ \bullet zero...) \bullet \underline{1} \\
&\overset{E-ISZEROSUCC}{\rightarrow} (recr\ \langle\rangle_{Nat}\ \bullet zero \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{Nat};\ r, c \bullet x \rightsquigarrow if\ isZero(x)\ then\ c\ else\ r \bullet x) \bullet \underline{1} \\
&\overset{E-RECR1}{\rightarrow} (if\ isZero(zero)\ then\ \langle\rangle_{Nat}\ else\ r \bullet x) \bullet \underline{1} \\
&\overset{E-ISZEROZERO}{\rightarrow} \langle\rangle_{Nat} \bullet \underline{1}
\end{aligned}
$$