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 declaratividad

1.2 Ejercicio 2

```haskell
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

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

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

  - Caso xs vacía

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

  - Caso xs no-vacía

$$esPreRama\ (Hoja\ x)\ xs \Rightarrow length\ xs \leq 1$$
$$\overset{\{E0\}}{\equiv} null\ xs\ ||\ \ (xs == [x]) \Rightarrow length\ xs \leq 1$$
$$\overset{\{N0\}}{\equiv} (xs == [x]) \Rightarrow length\ xs \leq 1$$

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 $P(i) \wedge P(d)$

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

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

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

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

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

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 Dicc(o, τ) representa diccionarios con claves de tipo o y valores de tipo τ.
- Vacío$_{o,\tau}$ es un diccionario vacío con claves de tipo o y valores de tipo τ.
- definir(M, N, O) define el valor O en el diccionario M para la clave N.
- def?(M, N) indica si la clave N fue definida en el diccionario M.
- 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 :

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

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

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

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

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

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

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

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 :

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

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

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

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

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

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

$\xrightarrow{E-IFTRUE} True$