Prácticas: Paradigmas de Lenguajes de Programación

Zamboni, Gianfranco

11 de febrero de 2018

Índice

0. Pre-Práctica de Programación Funcional

0.1. Ejercicio 1

`null :: Foldable t => t a -> Bool` indica si una estructura está vacía. El tipo a debe ser de la clase Foldable, esto es, son tipos a los que se les puede aplicar la función foldr. La notación ”t aïndica que es un tipo parámetrico, es decir, un tipo t que usa a otro tipo a, por ejemplo, si le pasamos a la función una lista de enteros, entonces a = Int y t = [Int]

`head :: [a] -> a` devuelve el primer elemento de una lista.

`tail :: [a] -> [a]` devuelve los últimos elementos de una lista (todos los elementos, salvo el primero).

`init :: [a] -> [a]` devuelve los primeros elementos de una lista (todos los elementos salvo el último).

`last :: [a] -> a` devuelve el último elemento de una lista.

`take :: Int -> [a] -> [a]` devuelve los primeros n elementos de una lista

`drop :: Int -> [a] -> [a]` devuelve los últimos n elementos de una lista

`(++) :: [a] -> [a] -> [a]` concatena dos listas

`concat :: Foldable t => t [a] -> [a]` concatena todas las listas de un contenedor de listas que soporte la operación foldr.

`(!!) :: [a] -> Int -> a` devuelve el elemento de una lista `l` que se encuentra en la `n`-ésima posición. La numeración comienza desde 0.

`elem :: (Eq a, Foldable t) => a -> t a -> Bool`: Dada una estructura T que soporta la operación foldr y que almacene elementos del tipo a que puedan ser comparados por medio de la igualdad y dado un elemento A de ese tipo, indica si A aparecen en T.

0.2. Ejercicio 2

```haskell
-- a) La función abs de Prelude ya hace esto
valorAbsoluto :: Float -> Float
valorAbsoluto x | x < 0     = -x
                | otherwise =  x

-- b)
bisiesto :: Int -> Bool
bisiesto x = (x `mod` 4) == 0

--c)
factorial :: Int -> Int
factorial 1 = 1
factorial x = x * factorial (x-1)

cantDivisoresPrimos :: Int -> Int
cantDivisoresPrimos x = length (filter esPrimo (divisores x))
```

```haskell
-- Auxiliares

esPrimo :: Int -> Bool
esPrimo x = length (divisores x) == 2

divisores :: Int -> [Int]
divisores x = [ y | y <- [1..x], x `mod` y == 0 ];
```

0.3. Ejercicio 3

```haskell
--a)
inverso :: Float -> Maybe Float
inverso 0 = Nothing
inverso x = Just (1/x)

-- b)
aEntero :: Either Int Bool -> Int
aEntero (Left x) = x
aEntero (Right x) | x == True = 1
                  | otherwise = 0
```

0.4. Ejercicio 4

```haskell
--a)
limpiar :: String -> String -> String
limpiar xs ys = [ y | y <- ys, not(elem y xs) ]

-- b)
difPromedio :: [Float] -> [Float]
difPromedio xs = map (\y -> y - promedio xs) xs
    where promedio xs = (sum xs) / (genericLength xs)

-- c)
todosIguales :: [Int] -> Bool
todosIguales =
        foldr (\y  rec -> ((length xs == 1) || (y == (head xs)))
                && rec) True
```

0.5. Ejercicio 5

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

-- a)
vacioAB:: AB a -> Bool
vacioAB Nil = True
vacioAB (Bin _ _ _) = False

-- b)
negacionAB :: AB Bool -> AB Bool
negacionAB Nil = Nil
negacionAB (Bin l x r) =
    Bin (negacionAB l) (not x) (negacionAB r)

-- c)
productoAB :: AB Int -> Int
productoAB Nil = 1
productoAB (Bin l x r) = x * (productoAB l) * (productoAB r)
```

1. Programación Funcional

Tipos en Haskell

1.1. Ejercicio 1

```haskell
-- La función max de Prelude ya hace esto
max2 ::(Float, Float) -> Float
max2 (x, y) | x >= y = x
            | otherwise = y

max2Currificada :: Float -> Float -> Float
max2Currificada x y | x >= y = x
                    | otherwise = y

normaVectorial :: (Float, Float) -> Float
normaVectorial (x, y) = sqrt (x^2 + y^2)

normaVectorial :: Float -> Float -> Float
normaVectorial x y = sqrt (x^2 + y^2)

-- subtract ys esta definida en Prelude
subtract1 :: Float -> Float -> Float
subtract1 = flip (-)

-- La función pred definida en Prelude ya hace esto
predecesor :: Float -> Float
predecesor = subtract 1

evaluarEnCero :: (Float -> b) -> b
evaluarEnCero = \f -> f 0

dosVeces :: (a -> a) -> (a -> a)
dosVeces = \f -> f.f

flipAll :: [a -> b -> c] -> [ b -> a -> c]
flipAll = map flip

flipRaro :: b -> ( a -> b -> c ) -> a -> c
flipRaro = flip flip
```

Listas por Compresión

1.2. Ejercicio 2

```haskell
[ x | x <- [1..3], y <- [x..3], ( x + y ) `mod` 3 == 0 ]
  = [ 1, 3 ]
```

1.3. Ejercicio 3

```haskell
pitagoricas :: [(Integer, Integer, Integer)]
pitagoricas =[(a, b, c) | a <- [1..],
                          b <- [1..],
                          c <- [1..],  a^2 + b^2 == c^2]
```

Esta definición agrega la tupla (1,1,1) a la lista y luego aumenta c infinitamente, sin encontrar ningun nueva coincidencia. Si cambiamos el orden en el que se recorren las listas y agregando algunas cotas de la siguiente forma:

```haskell
pitagoricas :: [(Integer, Integer, Integer)]
pitagoricas = [ (a, b, c) | c <- [1..],
                            b <- [1..c],
                            a <- [1..c], 2a^2 + b^2 == c^2]
```

En este caso, para cada número probamos todas las combinaciones de pares (a,b) tales que la suma de sus cuadrados podría llegar a dar c. Como a y b están acotados por c, ya que claramante c^2 + c^2 > c^2, la cantidad de pruebas de pares para cada número es finita (2^c pares) y es posible pasar al siguiente número una vez realizados estos chequeos.

1.4. Ejercicio 4

```haskell
primerosPrimos :: Int -> [Int]
primerosPrimos n = take n [ x | x <- [2..], esPrimo x ]
```

Gracias a la evaluación *lazy*, cuando se encuentran los primeros n primos la función deja de computar la lista de primos.

1.5. Ejercicio 5

```haskell
partir :: [a] -> [ ([a], [a]) ]
partir xs = [ (take i xs, drop i xs) | i <- [0..(length xs)] ]
```

1.6. Ejercicio 6

```haskell
listasQueSuman :: Int -> [[Int]]
listasQueSuman 1 = [[1]]
listasQueSuman n =
    [n]:( concat
        [ map ((n-i):) (listasQueSuman i) | i <- [ 1..n-1 ] ]
```

1.7. Ejercicio 7

```haskell
listasFinitas :: [[Int]]
listasFinitas = concat [ listasQueSuman i | i <- [1..]]
```

Currificación

1.8. Ejercicio 8

```haskell
-- I. curry y uncurry ya están definidas en Prelude
curry1 :: ((a,b) -> c) -> a -> b -> c
curry1 f a b = f (a,b)

-- II.
uncurry1 :: (a -> b -> c) -> (a, b) -> c
uncurry1 f (a, b) = f a b
```

III. No podemos definir una función `curryN` que tome una función con un número arbitrario de parametros, ya que la cantidad de parámetros de la función currificada depende de la cantidad de parámetros de la función original. Esto significa que `curryN` debería poder modificar la cantidad de parámetros que toma dependiendo de la función que se le pasa, lo que es imposible.

Otra idea sería tratar de definirla de manera que dada una función vaya remplazando los parámetros de a poco generando, de esta forma, n funciones parciales. Pero esto es imposible ya que la función debe tener la tupla de parámetros completa para poder ser evaluada de cualquier manera.

Esquemas de recursión

1.9. Ejercicio 9

```haskell
-- I.
dc :: DivideConquer a b
dc esTrivial resolver repartir combinar x =
    if esTrivial x then
        resolver x
    else combinar (map  dc1 (repartir x))
    where dc1 = dc esTrivial resolver repartir combinar

-- II.
mergesort :: Ord a => [a] -> [a]
mergesort = dc
    ((<=1).length)
    id
    partirALaMitad
    (\[xs,ys] -> merge xs ys)

--III.
mapDC :: (a -> b) -> [a] -> [b]
mapDC f = dc
    ((<=1).length)
    ( \xs -> if (length xs) == 0 then []
             else [ f (head xs) ] )
    partirALaMitad
    concat

filterDC :: (a -> Bool) -> [a] -> [a]
filterDC p = dc ((<=1).length)
    (\xs -> if (length xs == 0) || (p (head xs)) then []
                else xs )
    partirALaMitad
    concat

-- Auxiliares

partirALaMitad :: [a] -> [[a]]
partirALaMitad xs = [ take i xs, drop i xs ]
    where i = (div (length xs) 2)

merge :: Ord a => [a] -> [a] -> [a]
merge = foldr
    (\y rec -> (filter (<= y) rec) ++ [y] ++ (filter (>y) rec))
```

1.10. Ejercicio 10

```haskell
-- I.
sumFold :: Num a => [a] -> a
sumFold = foldr (+) 0

elemFold :: Eq a => a -> [a] -> Bool
elemFold x = foldr (\y rec -> (y==x) || rec) False

masMasFold :: [a] -> [a] -> [a]
masMasFold = flip (foldr (\x rec-> x:rec) )

mapFold :: (a->b) -> [a] -> [b]
mapFold f = foldr (\x rec-> (f x):rec) []

filterFold :: (a->Bool) -> [a] -> [a]
filterFold p = foldr (\x rec -> if (p x) then x:rec else rec) []
```

II. La función `foldr1 :: Foldable t => (a -> a -> a) -> t a -> a` está definida en Prelude. Esta función es una variante de `foldr` en la que el caso base se da cuando la estructura contiene un único elemento y ese elemento es el resultado del caso base.

```haskell
mejorSegun :: (a -> a -> Bool) -> [a] -> a
mejorSegun f xs =
    foldr1 (\x rec -> if f x rec then x else rec) xs

-- III.
sumaAlt :: Num a => [a] -> a    -- Preguntar
sumaAlt =  foldr (-) 0

-- IV.
sumaAlt2 :: Num a => [a] -> a
sumaAlt2 =  sumaAlt.reverse

-- V.
permutaciones :: [a] -> [[a]]
permutaciones = foldr
    (\x rec-> concatMap (agregarEnTodasLasPosiciones x) rec)
    [[]]
    where agregarEnTodasLasPosiciones j js =
        [ (fst h)++[j]++(snd h)| h <- (partir js)]
```

1.11. Ejercicio 11

```haskell
-- I.
partes :: [a] -> [[a]]
partes = foldr (\x res -> res ++ (map (x:) res)) [[]]

-- II.
prefijos :: [a] -> [[a]]
prefijos xs = [take i xs | i <- [0..(length xs)]]

-- III.
sublistas :: [a] -> [[a]]
sublistas xs = [[]] ++ [ take j (drop i xs)
        | i<-[0..(length xs)] , j<-[1..(length xs)-i]]
```

1.12. Ejercicio 12

```
-- a.
sacarUna :: Eq a => a -> [a] -> [a]
sacarUna x = recr (\y ys rec -> if (x==y) then ys else y:rec) []
```

b. `recr`, nos permite escribir funciones recursivas cuyo paso recursivo no solo dependen del paso anterior, sino que tambien dependen de la cola de la lista. Mientras que `foldr` es el esquema recursivo de inducción estructural, es decir nos permite definir funciones que solo dependen del caso anterior.

c. En cuanto a la función `listasQueSuman` del ejercicio 6, vemos que el valor de esta función depende de todos los casos anteriores, por lo que se hacen tantas llamadas recursivas como casos anteriores haya. Evidentemente, ni `fold` y ni `recr` nos dan un mecanismo para hacer esto.

1.13. Ejercicio 13

```
-- I.
genLista :: a -> (a -> a) -> Int -> [a]
genLista x proximo n =
    foldr (\x rec ->  if null rec then [x]
                      else rec ++ [ proximo (last rec)])
          []
          [1.. n ]

-- II.
desdeHasta :: Int -> Int -> [Int]
desdeHasta x z = genLista x (+1) (z-x)
```

1.14. Ejercicio 14

```
-- I.
mapPares :: (a -> b -> c) -> [(a,b)] -> [c]
mapPares f = map (unCurry f) xs

-- II.
armarPares :: [a] -> [b] -> [(a,b)]
armarPares xs ys =
    if (length xs) > (length ys) then
       foldr
           (\x rec-> \ys -> (x,head ys):(rec (tail ys)) )
           (\ys -> [])
           xs ys
    else
        foldr (\y rec-> \xs -> (head xs, y):(rec (tail xs)) )
        (\xs -> [])
        ys xs

--III.
mapDoble :: (a -> b -> c) -> [a] -> [b] -> [c]
mapDoble f as  = mapPares f.(zip as)
```

1.15. Ejercicio 15

```
-- I.
sumaMat :: [[Int]] -> [[Int]] -> [[Int]]
sumaMat = zipWith (zipWith (+))

- II.
trasponer :: [[Int]] -> [[Int]]
trasponer [] = []
trasponer xss = foldr (zipWith (:))
    [ [] | i <- [1..length (head xss)]] xss
```

1.16. Ejercicio 16

```
generateBase::([a] ->Bool) ->a ->(a ->a) ->[a]
generateBase stop x next =
    generate stop
             (\xs -> if ((length xs) == 0) then x
                     else (next (last xs)) )

-- I.
generateBase::([a] ->Bool) ->a ->(a ->a) ->[a]
generateBase stop x next =
    generate stop
        (\xs -> if ((length xs) == 0) then x
                else (next (last xs)) )

-- II.
factoriales::Int ->[Int]
factoriales n =
    generate ((==n).length)
             (\xs -> if null xs then 1
                     else (last xs)*(length xs))

-- III.
iterateN :: Int -> (a -> a) -> a -> [a]
iterateN n f x = generateBase ((>n).length) x f
```

IV. La función `iterate :: (a -> a) -> a -> [a]` toma una función f que calcula el proximo elemento de la lista basandose en el último elemento agregado y un valor inicial x y crea una lista infinita. La función `takeWhile :: (a -> Bool) -> [a] -> [a]` toma un predicado p y una lista l y devuelve elementos de la lista mientras cumplan p.

```
generateFrom1:: ([a] -> Bool) -> ([a] -> a) -> [a] -> [a]
generateFrom1 stop next =
    last.
    (takeWhile (not.stop)).
    (iterate (\ys -> ys ++ [next ys]))
```

Otras estructuras de datos

1.17. Ejercicio 17

```
-- I.
foldNat :: (Integer -> a -> a) -> a -> Integer -> a
foldNat _ z 0 = z
foldNat f z n = f n (foldNat f z (n-1))
```

```haskell
-- II.
potencia :: Integer -> Integer -> Integer
potencia n m = foldNat (\x -> (n*)) n (m-1)
```

1.18. Ejercicio 18

```haskell
type Conj a = (a->Bool)

-- I.
vacio :: Conj a
vacio x = False

agregar :: Eq a => a -> Conj a -> Conj a
agregar x c = (\y -> (y == x) || (c x))

-- II.
interseccion :: Conj a -> Conj a -> Conj a
interseccion c1 c2 = ( \x -> (c1 x) && (c2 x) )

union :: Conj a -> Conj a -> Conj a
union c1 c2 =( \x -> (c1 x) || (c2 x) )

-- III.
conjuntoInfinito :: Conj a
conjuntoInfinito = (\x -> True)

-- IV.
singleton :: Eq a => a -> Conj a
singleton x = (==x)
```

V. A diferencia de otros tipos, el tipo `Conj` está definido como una función booleana que depende de un único parametro. Dado un conjunto C, no tenemos formar de saber cuales son los elementos que contiene sin haber probado uno por uno cada uno de los elementos de su dominio. Por ejemplo, si tuviesemos un conjunto de enteros, entonces para poder aplicar una función a cada uno de sus elementos deberíamos probar la pertenencia para cada entero posible y aplicar la función a aquellos que estén en el conjunto, sin embargo, los enteros son infinitos, por lo que la función `map` nunca terminaría.

Teniendo esto en cuenta, podemos definir

```haskell
mapConjunto :: [a] -> (a -> b) -> Conj a -> Conj b
```

como una función que, dado el dominio $Dom(C)$ del conjunto C, un conjunto C y una función f, devuelve una función g parcialmente computable de tipo `b -> Bool`. Esta función g , dado `e::b` evaluará a `True` si y solo si existe $x \in Dom(C)$ tal que $f(x) = e$ y $x \in C$, es decir, si e pertenece al conjunto que devuelve `mapConjunto`. Ahora, si e no pertenece al conjunto, entonces pueden pasar dos cosas:

1. Si $Dom(C)$ es finito, entonces g evalúa a $false$.
2. Sino (si $Dom(C)$ es infinito) g se cuelga y nunca devuelve nada.

```haskell
mapConjunto :: Eq b => [a] -> (a -> b) -> Conj a -> Conj b
mapConjunto xs f c =
    (\x -> not (null (filter
                            (\y -> (c y) && ((f y) == x))
                            xs
                       )
                 )
    )
```

1.19. Ejercicio 19

Mostramos dos posibles implementaciones para las funciones `fila` y `columna`

```
-- I
fila :: Int -> MatrizInfinita a -> [a]
fila x m = [ m x i | i <- [1..]]

columna :: Int -> MatrizInfinita a -> [a]
columna x m = [ m i x | i <- [1..]]

-- II.
trasponerInfinito :: MatrizInfinita a -> MatrizInfinita a
trasponerInfinito m = (flip m)

-- III.
mapMatriz :: ( a -> b ) -> MatrizInfinita a -> MatrizInfinita b
mapMatriz f m = (\x y -> f (m x y))

filterMatriz :: ( a -> Bool ) -> MatrizInfinita a -> [a]
filterMatriz p m = [ m j (i-j) | i <-[0..], j <-[0..i], p (m j
↪  (i-j)) ]

zipWithMatriz :: (a->b->c) -> MatrizInfinita a
                 -> MatrizInfinita b -> MatrizInfinita c
zipWithMatriz f m1 m2 = (\i j -> f (m1 i j) (m2 i j))

--IV.
sumaMatriz :: Num a => MatrizInfinita a -> MatrizInfinita a
              -> MatrizInfinita a
sumaMatriz = zipWithMatriz (+)

zipMatriz :: MatrizInfinita a -> MatrizInfinita b
             -> MatrizInfinita (a,b)
zipMatriz = zipWithMatriz (\x y -> (x,y))
```

1.20. Ejercicio 20

Renombro el contructor `Bin` a `BinAHD` para que no rompa con el constructor de árboles binarios definidos en la práctica 0.

```
data AHD a b =
     Hoja b
   | Rama tInt (AHD a b)
   | BinAHD (AHD a b) a (AHD a b)

-- I.
foldAHD :: (a -> c -> c -> c) -> (a -> c -> c) -> (b -> c)
           -> AHD a b -> c
foldAHD _ _ z (Hoja e) = z e
foldAHD f g z (Rama e ar) = g e (foldAHD f g z ar)
foldAHD f g z (BinAHD ar1 e ar2) =
    f e (foldAHD f g z ar1) (foldAHD f g z ar2)
```

```haskell
-- II.
mapAHD :: (a -> b) -> (c -> d) -> AHD a c -> AHD b d
mapAHD f g =
foldAHD (\e rec1 rec2 -> BinAHD rec1 (f e) rec2)
        (\e rec -> Rama (f e) rec)
        (\e -> Hoja (g e))
```

1.21. Ejercicio 21

```haskell
-- I
foldAB :: (a -> b -> b -> b) -> b -> AB a -> b
foldAB _ z Nil = z
foldAB f z (Bin ar1 e ar2) =
    f e (foldAB f z ar1) (foldAB f z ar2)

-- II
esNil :: AB a -> Bool
esNil arbol =
    case arbol of
               Nil -> True
               Bin _ _ _ -> False

altura :: AB a -> Integer
altura = foldAB (\x rec rec1 -> 1 + rec + rec1) 0

ramas :: AB a -> [[a]]
ramas = foldAB
    (\e rec rec1 -> if (null rec) && (null rec1) then [[e]]
                    else map (e:) (rec++rec1)
    )
    []

nodos :: AB a -> Integer
nodos = foldAB (\_ rec rec1 -> 1 + rec + rec1) 0

hojas :: AB a -> Integer
hojas =
    foldAB (\_ rec rec1 -> if (rec == 0) && (rec1 == 0) then 1
                           else rec + rec1
           )
           0

espejo :: AB a -> AB a
espejo = foldAB (\x rec rec1 -> Bin rec1 x rec)
                Nil
```

```haskell
-- III
root :: AB a -> a
root (Bin _ x _) = x

izq :: AB a -> AB a
izq (Bin i _ _) = i

der :: AB a -> AB a
der (Bin _ _ d) = d

mismaEstructura :: Eq a => AB a -> AB a -> Bool
mismaEstructura =
    foldAB  (\x rec rec1 ->
        (\arbol ->
           if esNil arbol then False
           else (root arbol) == x &&
                (rec (izq arbol)) &&
                (rec1 (der arbol))
        )
        (\arbol -> esNil arbol)
```

1.22. Ejercicio 22

```haskell
-- I.
data RoseTree = Rose a [RoseTree a]

-- II.
foldRose :: (a -> [b] -> b) -> RoseTree a -> b
foldRose f (Rose x xs) = f x (map (foldRose f) xs)

-- III.a)
hojasRT :: RoseTree a -> [a]
hojasRT = foldRose (\x recs -> if null recs then [x]
                               else  x: (concat recs))
--- III.b)
distancias :: RoseTree a -> [(a, Integer)]
distancias =
    foldRose (\x recs -> if null recs then [(x,0)]
                         else map (\(t1,t2) -> (t1, t2 + 1))
                                  (concat recs))
-- III.c)
alturaRT :: RoseTree a -> Integer
alturaRT = foldRose (\x recs -> if null recs then 0
                         else 1 + (max recs))
    where max = mejorSegun (>)
```

2. Introducción al Cálculo Lambda Tipado

Sintaxis

2.1. Ejercicio 1

En este ejercicio, nos piden identificar las expresiones sitacticamente válidas. Tenemos que tener cuidado de no confundir estas expresiones con las expresiones correctamente tipadas. Todas las expresiones que nos permite escribir el conjunto de términos son expresiones válidas sintacticamente, aún si estas no pueden ser tipadas.

Expresiones de términos	**Expresiones de tipo**
x	σ
M	$Bool$
$x\ x$	$Bool \to Bool$
$M\ M$	$Bool \to Bool \to Bool$
$true\ false$	$Bool \to Bool \to Nat$
$true\ succ(true\ false)$	$(Bool \to Bool) \to Nat$
$\lambda x : \sigma.succ(x)$	
$\lambda x : Bool.succ(x)$	
$\lambda x : Bool.if\ 0\ then\ true\ else\ 0\ succ(x)$	

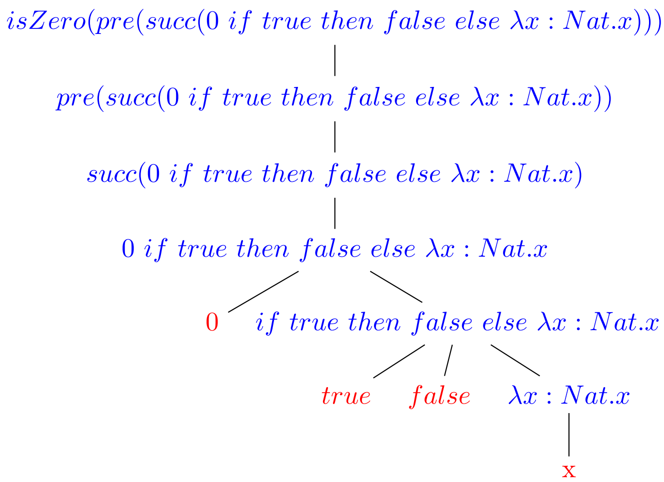
No validas

$\lambda x.isZero(x)$

$\lambda x : if\ true\ then\ Bool\ else\ Nat.x$

$succ\ true$

2.2. Ejercicio 2



2.3. Ejercicio 3

$$\lambda x : Nat.succ((\lambda x : Nat.x)\ x)$$

En el término $\lambda x_1 : Nat.succ(x_2)$, x_1 no aparece como subtérmino.]

2.4. Ejercicio 4

A la espera...

Tipado

2.5. Ejercicio 5

$$\dfrac{\dfrac{}{\phi \triangleright true : Bool}\,\text{T-True} \qquad \dfrac{}{\phi \triangleright 0 : Nat}\,\text{T-Zero} \qquad \dfrac{\dfrac{}{\phi \triangleright 0 : Nat}\,\text{T-Zero}}{\phi \triangleright succ(0) : Nat}\,\text{T-Succ}}{\phi \triangleright if\ true\ then\ 0\ else\ succ(0) : Nat}$$

En la siguiente demostración $\Gamma = \{x : Nat,\ y : Bool\}$

$$\dfrac{\dfrac{}{\Gamma \triangleright true : Bool}\,\text{T-True} \qquad \dfrac{}{\Gamma \triangleright false : Bool}\,\text{T-False} \qquad \dfrac{\dfrac{\dfrac{z : Bool \in \Gamma, z : Bool}{\Gamma, z : Bool \triangleright z : Bool}\,\text{T-Var}}{\Gamma \triangleright \lambda z : Bool.z : Bool \rightarrow Bool}\,\text{T-Abs} \qquad \dfrac{}{\Gamma \triangleright true : Bool}\,\text{T-True}}{\Gamma \triangleright (\lambda z : Bool.z)\ true : Bool}\,\text{T-App}}{\Gamma \triangleright if\ true\ then\ false\ else\ (\lambda z : Bool.z)\ true : Bool}\,\text{T-If}$$

$$\dfrac{\dfrac{\color{red}{\phi, x : Bool \triangleright x : Bool \Rightarrow \phi \triangleright \lambda x : Bool.x : Bool \rightarrow \tau}}{\phi \triangleright \lambda x : Bool.x : Bool}\,\text{T-Abs} \qquad \phi \triangleright 0 : Nat \qquad \phi \triangleright succ(0) : Nat}{\phi \triangleright if\ \lambda x : Bool.x\ then\ 0\ else\ succ(0) : Nat}\,\text{T-If}$$

En la próxima demostración $\Gamma = \{x : Bool \to Nat, y : Bool\}$

$$\dfrac{\dfrac{x : Bool \to Nat \in \Gamma}{\Gamma \triangleright x : Bool \to Nat}\text{T-Var} \qquad \dfrac{y : Bool \in \Gamma}{\Gamma \triangleright y : Bool}\text{T-Var}}{\Gamma \triangleright if\ \lambda x : Bool.x\ then\ 0\ else\ succ(0) : Nat}\text{T-App}$$

2.6. Ejercicio 6

$$\dfrac{\dfrac{}{\phi \triangleright 0 : Nat}\text{T-Zero}}{\phi \triangleright succ(0) : Nat}\text{T-Succ} \qquad \dfrac{\dfrac{}{\phi \triangleright succ(0) : Nat}\text{Ya demostrado}}{\phi \triangleright isZero(succ(0)) : Bool}\text{T-Zero}$$

$$\dfrac{\dfrac{\dfrac{}{\phi \triangleright true : Bool}\text{T-True} \quad \dfrac{}{\phi \triangleright false : Bool}\text{T-True} \quad \dfrac{}{\phi \triangleright false : Bool}\text{T-True}}{\Gamma \triangleright if\ true\ then\ false\ else\ false : Bool}\text{T-If} \quad \dfrac{}{\phi \triangleright 0 : Nat}\text{T-Zero} \quad \dfrac{}{\phi \triangleright succ(0) : Nat}\text{Ya demostrado}}{\phi \triangleright if\ if\ true\ then\ false\ else\ false\ then\ 0\ else\ succ(0) : Nat}\text{T-If}$$

2.7. Ejercicio 7

En la próxima demostración $\Gamma = \{x : \sigma\}$

$$\dfrac{\dfrac{\dfrac{x : Nat \in \Gamma}{\Gamma \triangleright x : Nat}\text{T-Var}}{\Gamma \triangleright succ(x) : Nat}\text{T-Succ}}{\Gamma \triangleright isZero(succ(x)) : \tau}\text{T-Succ}$$

Entonces, para que la demostración tenga sentido debe pasar que $\sigma = Nat$ y $\tau = Bool$

$$\dfrac{\dfrac{\{x : \sigma\} \triangleright x : \sigma}{\phi \triangleright \lambda x : \sigma.x : \sigma \to \sigma}\text{T-Abs} \qquad \dfrac{\{y : Bool\} \triangleright 0 : Nat}{\phi \triangleright \lambda y : Bool.0 : \sigma}\text{T-Abs}}{\phi \triangleright (\lambda x : \sigma.x)\ (\lambda y : Bool.0) : \sigma}\text{T-App}$$

El árbol de la segunda abstracción nos dice que $\sigma = Bool \to Nat$ y como el desgloce de la primera abstracción, no impuso ninguna otra condición sobre σ, entonces con este tipo funciona.