	Escuela de Ingeniería	
Examen de Fundamentos de la Computación		**Código de materia: 6449**
Fecha: 19/10/2021		**Hoja 1 de 3**
Duración: 2 horas	**Examen**	**Sin material**

Fundamentos de la Computación
Examen

A no ser que la letra indique lo contrario, toda función que se use debe ser definida (lo cual incluye declarar su tipo), y todo lema utilizado debe ser demostrado.

Problema 1. [35p]

(a) Defina la función `zipWith :: (a->b->c) -> [a] -> [b] -> [c]` que recibe una función y dos listas, y construye una lista partir de aplicar la función a los elementos de las listas de entrada que ocurren en la misma posición en ambas listas. Si una lista es más larga que otra, no se consideran los elementos sobrantes.
Ejemplos: `zipWith (>) [1,7,2] [2,3,4,5] = [False,True,False]`

(b) Demuestre que `(∀f::a->b->c)(∀l1::[a])(∀l2::[b])length (zipWith f l1 l2)` $\leq$ `length l1`, donde `length :: [a] -> N` se define como:
`length = \l -> case l of {[] -> 0 ; x:xs -> S (length xs)}`.
Puede asumir, sin necesidad de demostrarlas, las siguientes propiedades de $\leq$:

L1: `(∀n::N) 0` $\leq$ `n`

L2: `(∀n::N) n` $\leq$ `n`

L3: `(∀m,n::N) m` $\leq$ `n` $\Rightarrow$ `S m` $\leq$ `S n`

Solución

(a)
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith = \f l1 l2 -> case l1 of {[] -> [];
                                  x:xs -> case l2 of {[] -> [];
                                                      z:zs -> f x z :  zipWith f xs zs }}
```

(b) `(∀f::a->b->c)(∀l1::[a])(∀l2::[b])length (zipWith f l1 l2)` $\leq$ `length l1`.
Dem. Sea `f::a->b->c`. Por inducción en `l1::[a]`
Caso `l = []`: `(∀l2::[b])length (zipWith f [] l2)` $\leq$ `length []`.
Dem. Sea `l2::[b]`.
`length (zipWith f [] l2)`
= (def. `zipWith`)
`length []`
= (def. `length`)
`0`
$\leq$ (L1)
`length []`

Caso `l1 = x:xs`, con `x::a` y `xs::[a]` cualesquiera.
HI1) `(∀l2::[b])length (zipWith f xs l2)` $\leq$ `length xs`
TI1) `(∀l2::[b])length (zipWith f (x:xs) l2)` $\leq$ `length (x:xs)`
Dem. Por inducción en `l2::[b]`.
Caso `l2 = []`: `(∀l2::[b])length (zipWith f (x:xs) [])` $\leq$ `length (x:xs)`
Dem.
`length (zipWith f (x:xs) [])`

= (def. `zipWith`)
`length []`
= (def. `length`)
`0`
$\leq$ (L1)
`length (x:xs)`

Caso `l2 = z:zs`, con `z::b` y `zs::[b]` cualesquiera.

HI2) `length (zipWith f (x:xs) zs)` $\leq$ `length (x:xs)`

TI2) `length (zipWith f (x:xs) (z:zs))` $\leq$ `length (x:xs)`

Dem.
`length (zipWith f (x:xs) (z:zs))`
= (def. `zipWith`)
`length (f x z :  zipWith f xs zs`
= (def. `length`)
`S (length (zipWith f xs zs))`
$\leq$ (L3, con HI1 aplicada al caso `l2 = zs`)
`S(length xs)` = (def. `length`)
`length (x:xs)`

Problema 2. `[65p.]`

Considere la siguiente definición:

```
f :: T a b -> a -> b -> Bool
f = \t v w -> case t of { X i j -> case i of { 0 -> True ; S x -> not j};
                          Y k l m -> f m l k;
                          Z p q r -> case r of {[] -> f p v w ; s:ss -> f q v s }
```

(a) Defina el tipo `T a b` para que la función `f` compile:
`data T a b where {X ::...  ; Y ::...  ; Z ::...  }.`

(b) Defina la función `cantb :: T a b -> N`, que recibe un árbol de tipo `T a b` y calcula la cantidad de elementos de tipo `b` que tiene.
Puede utilizar la suma de naturales `(+) :: N -> N -> N` y la función `length :: [a] -> N` definida en el ejercicio anterior.

(c) Defina la función `mapb :: (b->b) -> T a b -> T a b`, que recibe una función `f` de tipo `b->b` y un árbol `t` de tipo `T a b`, y le aplica `f` a todos los elementos de tipo `b` del árbol.
Puede utilizar la función `map` de listas, definida como:

```
map ::  (a -> b) -> [a] -> [b]
map = \f l -> case l of {[] -> [] ; x:xs -> f x :  map f xs}
```

(d) Demuestre que (∀`f::b->b`)(∀`t::T a b`) `cantb t = cantb (mapb f t)`
Puede asumir, sin necesidad de demostrarlos, la asociatividad de la suma y la siguiente propiedad de `length`: (∀`f::a->b`)(∀`xs::[a]`) `length xs = length (map f xs)`

Solución

(a)
```
data T a b where { X ::  N -> Bool -> T a b;
                   Y ::  b -> a -> T a b -> T a b;
                   Z ::  T a b -> T a b -> [b] -> T a b }
```

(b)
```
cantb :: T a b -> N
cantb = \t -> case t of { X i j -> 0;
                          Y k l n -> S(cantb n);
                          Z p q r -> cantb p + cantb q + length r }
```

(c)
```
mapb ::(b ->b) -> T a b -> T a b
mapb = \f t -> case t of { X i j -> X i j;
                           Y k l n -> Y (f k) l (mapb f n);
                           Z p q r -> Z (mapb f p) (mapb f q) (map f r) }
```

(d) $(\forall$`f::a -> b`$)(\forall$`t::T a b`$)$ `cantb t = cantb (mapb f t)`

Dem. Sea `f::a -> b`. Por inducción en `t::T a b`

Caso `t = X i j`, con `i::N, j::Bool` cualesquiera: `cantb(X i j) = cantb(mapb f (X i j))`

Dem.
`cantb (X i j) = cantb (mapb f (X i j))`
⇔ (def `cantb` ; def `mapb`)
`0 = cantb (X i j)`
⇔ (; def `cantb`)
`0 = 0`
Se cumple por reflexividad del =.

Caso `t = Y k l n`, con `k::b, l::a, n::T a b` cualesquiera.
HI) `cantb n = cantb (mapb f n)`
TI) `cantb (Y k l n) = cantb (mapb f (Y k l n))`

Dem.
`cantb (Y k l n) = cantb (mapb f (Y k l n))`
⇔ (def `cantb` ; def `mapb`)
`S(cantb n) = cantb (Y (f k) l (mapb f n))`
⇔ (; def `cantb`)
`S(cantb n) = S(cantb (mapb f n))`
⇔ (HI)
`S(cantb n) = S(cantb n)`
Se cumple por reflexividad del =.

Caso `t = Z p q r`, con `p::T a b, q::T a b, r::[b]` cualesquiera.
HI1) `cantb p = cantb (mapb f p)`
HI2) `cantb q = cantb (mapb f q)`
TI) `cantb (Z p q r) = cantb (mapb f (Z p q r))`

Dem. Asumimos asociatividad de +, por lo que omitimos los paréntesis en las sumas.
`cantb (Z p q r) = cantb (mapb f (Z p q r))`
⇔ (def `cantb` ; def `mapb`)
`cantb p + cantb q + length r = cantb (Z (mapb f p) (mapb f q) (map f r))`
⇔ (; def `cantb`)
`cantb p + cantb q + length r = cantb(mapb f p) + cantb(mapb f q) + length(map f r)`
⇔ (HI1 y HI2)
`cantb p + cantb q + length r = cantb p + cantb q + (length(map f r))`
⇔ (Lema de letra)
`cantb p + cantb q + length r = cantb p + cantb q + length r`
Se cumple por reflexividad del =.