| | |
|---|---|
| **Escuela de Ingeniería** | |
| **Examen de Fundamentos de la Computación** | **Código de materia: 6449** |
| **Fecha: 05/03/2021** |  |
| **Duración: 2,5 horas** | **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.** `[32p.]`

(a) Defina sin usar funciones auxiliares, la función `opuestos :: [Bool] -> [Bool]` que recibe una lista de booleanos y devuelve otra, que se obtiene cambiando cada valor de la lista recibida por su opuesto.
Ejemplo: `opuestos [True, False, True] = [False, True, False]`

(b) Defina la función `falses :: [Bool] -> N` que recibe una lista de booleanos, y cuenta la cantidad de `False` que aparecen en ella.
Ejemplo: `falses [True, False, False] = S (S O)`

(c) Demuestre que (∀`l::[Bool]`) `falses l + falses (opuestos l) = length l`, donde
`(+) :: N -> N -> N` es la suma de naturales, definida como
`(+) = \x y -> case x of {0 -> y ; S z -> S(z+y)}`, y
`length :: [a] -> N` se define como
`length = \l -> case l of {[ ] -> 0 ; x:xs -> S(length xs)}`.
También puede asumir la conmutatividad de la suma sin necesidad de demostrarla, o sea:
(∀ `x,y::N`) `x + y = y + x`.

**Solución**

(a)
```
opuestos :: [Bool] -> [Bool]
opuestos = \l -> case l of { [ ] -> [ ];
                             (y:ys) -> case y of {
                                         True -> False:(opuestos ys);
                                         False -> True:(opuestos ys)}
                                         }
                           }
```

(b)
```
falses :: [Bool] -> N
falses = \l -> case l of { [ ] -> O;
                           (y:ys) -> case y of {
                                       True -> falses ys;
                                       False -> S(falses ys)}
                                       }
                         }
```

(c) (∀`l::[Bool]`) `falses l + falses (opuestos l) = length l`.
Dem. Por inducción en `l::[Bool]`
**Caso** [ ]: `falses [ ] + falses (opuestos [ ]) = length [ ]`
⟺ (def. `opuestos`)
`falses [ ] + falses [ ] = (length [ ])`

$\Longleftrightarrow$ (def. `falses` x 2; def. `(+)`; def. `length`)
`0 = 0`
Lo cual se cumple por reflexividad del =.

**Caso (:):** Sea `xs::[Bool]`
HI) `falses xs + falses (opuestos xs) = length xs`
TI) ($\forall$`x::Bool`) `falses (x:xs) + falses (opuestos (x:xs)) = length (x:xs)`
Dem. Por casos en `x::Bool`

**Caso True:**
`falses (True:xs) + falses (opuestos (True:xs)) = length (True:xs)`
$\Longleftrightarrow$ (def. `opuestos`)
`falses (True:xs) + falses (False:(opuestos xs)) = length (True:xs)`
$\Longleftrightarrow$ (def. `falses` x 2)
`falses xs + S(falses (opuestos xs)) = S(length xs)`
$\Longleftrightarrow$ (conmutatividad de +)
`S(falses (opuestos xs)) + falses xs = S(length xs)`
$\Longleftrightarrow$ (def.+)
`S(falses xs + falses (opuestos xs)) = S(length xs)`
$\Longleftrightarrow$ (HI)
`S(length xs) = S(length xs)`
Lo cual se cumple por reflexividad del =.

**Caso False:**
`falses (False:xs) + falses (opuestos (False:xs)) = length (False:xs)`
$\Longleftrightarrow$ (def. `opuestos`)
`falses (False:xs) + falses (True:(opuestos xs)) = length (False:xs)`
$\Longleftrightarrow$ (def. `falses`)
`S(falses xs) + falses (opuestos xs) = S(length xs)`
$\Longleftrightarrow$ (def.+)
`S(falses xs + falses (opuestos xs)) = S(length xs)`
$\Longleftrightarrow$ (HI)
`S(length xs) = S(length xs)`
Lo cual se cumple por reflexividad del =.

**Problema 2.** [30p.]

Considere la siguiente definición:

```
f ::  N -> B b -> b ->[b]
f = \x y z -> case y of { A i j k -> case j of { O -> [i] ; S n -> [k] };
                          B l m n o -> case l of { True -> f x m z ;
                                                   False -> f (fst n) o (snd n) };
                          C p q r -> p : f r q z }}
```

(a) Defina el tipo `B b` para que la función `f` compile:
`data B b where {A ::... ; B ::... ; C ::...}`, sabiendo que
```
fst :: (a,b) -> a
fst = λ p -> case p of { (x,y) -> x }
snd :: (a,b) -> b
snd = λ p -> case p of { (x,y) -> y }
```

(b) Defina la función `cantHojas :: B b -> N` que recibe un árbol y calcula la cantidad de hojas que tiene.

(c) Defina la función `sumar :: B b -> N` que recibe un árbol y devuelve la suma de todos los naturales que tiene (asumiendo que `b` $\neq$ `N`).

Puede utilizar la `suma` de naturales definida arriba y las funciones `fst` y `snd`.

**Solución**

(a)
```
data B b  where { A :: b -> N -> b -> B b  ;
                  B :: Bool -> B b  -> (N,b)  -> B b  -> B b  ;
                  C :: b -> B b  -> N -> B b }
```

(b)
```
cantHojas :: B b -> N
cantHojas = \t -> case t of { A i j k -> S 0;
                              B l m n o -> cantHojas m + cantHojas o;
                              C p q r -> cantHojas q}
```

(c)
```
sumar :: B b -> N
sumar = \t -> case t of { A i j k -> j;
                          B l m n o -> sumar m + fst n + sumar o;
                          C p q r -> sumar q + r }
```

**Problema 3.** [30p.]

Considere la siguiente definición:

```
f ::  N -> B b -> b ->[b]
f = \x y z -> case y of { A i j k -> case j of { 0 -> [i] ; S n -> [k] };
                          B l m n o -> case l of { True -> f x m z ;
                                                   False -> f (fst n) o (snd n) };
                          C p q r -> p : f r q z }}
```

(a) Defina el tipo `B b` para que la función `f` compile:
`data B b where {A ::...  ; B ::...  ; C ::...}`, sabiendo que
```
fst :: (a,b) -> a
fst = λ p -> case p of { (x,y) -> x }
snd :: (a,b) -> b
snd = λ p -> case p of { (x,y) -> y }
```

(b) Defina la función `cantHojas :: B b -> N` que recibe un árbol y calcula la cantidad de hojas que tiene.

(c) Defina la función `sumar :: B b -> N` que recibe un árbol y devuelve la suma de todos los naturales que tiene (asumiendo que `b` $\neq$ `N`).

Puede utilizar la `suma` de naturales definida arriba y las funciones `fst` y `snd`.

**Solución**

(a)
```
data B b  where { A :: b -> N -> b -> B b  ;
                  B :: Bool -> B b  -> (N,b)  -> B b  -> B b  ;
                  C :: b -> B b  -> N -> B b }
```

(b)
```
cantHojas :: B b -> N
cantHojas = \t -> case t of { A i j k -> S 0;
                              B l m n o -> cantHojas m + cantHojas o;
                              C p q r -> cantHojas q}
```

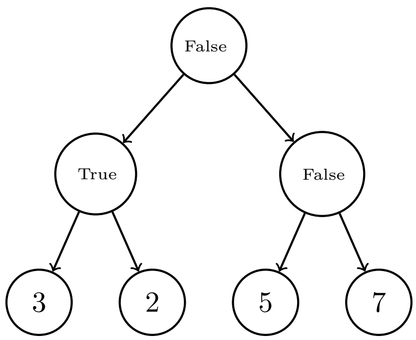
(c)
```
sumar :: B b -> N
sumar = \t -> case t of { A i j k -> j;
                          B l m n o -> sumar m + fst n + sumar o;
                          C p q r -> sumar q + r }
```

**Problema 4.** [38p.]

Considere el siguiente tipo `T a b` de árboles con nodos internos binarios que contienen información de tipo `b` y hojas que contienen información de tipo `a` :

```
data T a b where { H  ::  a -> T a b;
                   N  ::  T a b -> T a b -> b -> T a b}
```

(a) Codifique el siguiente árbol como una expresión `e`, y dé su tipo:




(b) Defina la función `mapIf`, que recibe un predicado `p`, una función `f` y un árbol `t` y le aplica `f` solamente a las `hojas` de `t` que cumplan `p`.
Cuál debe ser el tipo de `mapIf` para que la función compile?

(c) Defina la función `listarHojas :: T a b -> [a]` que recibe un árbol `t` y devuelve todas sus hojas en una lista. Puede hacer uso de la concatenación de listas:
`(++) :: N -> N -> N`, definida como
`(++) = \l1 l2 -> case l1 of { [] -> l2; x:xs -> x:(xs++l2)}`

(d) Demuestre por inducción:
`(∀t::T a b)(∀p::a -> Bool) listarHojas (mapIf p id t) = listarHojas t`, sabiendo que `id :: a -> a`, está definida como `id = \x -> x`.

**Solución**

(a)
```
e :: T Int Bool
e = N (N (H 3) (H 2) True) (N (H 5) (H 7) False) False
```

(b)
```
mapIf :: (a -> Bool) -> (a -> a) -> T a b -> T a b
mapIf = \p f t -> case t of { H x -> case p x of { True -> H (f x);
                                                   False -> H x};
                              N i d x -> N (mapIf p f i) (mapIf p f d) x}
```

(c)
```
listarHojas :: T a b -> [a]
listarHojas = \t -> case t of {H x -> [x];
                               N i d e -> (listarHojas i)++(listarHojas d)}
```

(d) `(∀t::T a b)(∀p::a -> Bool)listarHojas (mapIf p id t) = listarHojas t`
Dem. por inducción en `t::T a b`.

Caso H: $(\forall$`x::a`$)(\forall$`p::a -> Bool`) `listarHojas (mapIf p id (H x)) = listarHojas (H x)`

Dem. por casos en `p x::Bool`.

Caso True:

`listarHojas (mapIf p id (H x))`
= (def. `mapIf`)
listarHojas (H (id x))
= (def. `listarHojas`)
[id x]
= (def. `id`)
[x]
= (def. `listarHojas`)
`listarHojas (H x)`

Caso False:

`listarHojas (mapIf p id (H x))`
= (def. `mapIf`)
listarHojas (H x)

Caso N: Sean `i,d::T a b`

H1) $(\forall$`p::a -> Bool`) `listarHojas (mapIf p id i) = listarHojas i`

H2) $(\forall$`p::a -> Bool`) `listarHojas (mapIf p id d) = listarHojas d`

T)$(\forall$`x::b`$)(\forall$`p::a -> Bool`) `listarHojas (mapIf p id (N i d x) = listarHojas (N i d x)`

`listarHojas (mapIf p id (N i d x)`
= (def. `mapIf`)
`listarHojas (N (mapIf p id i) (mapIf p id d) x)`
= (def. `listarHojas`)
`listarHojas (mapIf p id i) ++ listarHojas (mapIf p id d)`
= (H1,H2)
`listarHojas i ++ listarHojas d`
= (def. `listarHojas`)
listarHojas `(N i d x)`