Escuela de Ingeniería

Examen de Fundamentos de la Computación — **Código de materia: 6449**

Fecha: 6/8/2020 — **Hoja 1 de 4**

Duración: 3 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. `[36p]`

(a) Defina la función `dejar :: N -> [a] -> [a]`, tal que `dejar n l` devuelve una lista con los primeros `n` elementos de la lista `l`. Si `l` tiene menos de `n` elementos, se debe devolver la lista `l`.
Ejemplo: `dejar (S(S 0)) "abcdef" = "ab"`.

(b) Defina la función `sacar :: N -> [a] -> [a]`, tal que `sacar n l` devuelve la lista que se obtiene sacando los primeros `n` elementos de la lista `l`. Si `l` tiene menos de `n` elementos, se debe devolver la lista vacía.
Ejemplo: `sacar (S(S 0)) "abcdef" = "cdef"`.

(c) Demuestre que `(∀n::N)(∀m::N)(∀l::[a]) sacar n (dejar (n+m) l) = dejar m (sacar n l)`, donde `(+) :: N -> N -> N` es la suma de naturales definida como:
`(+) = \x y -> case x of {0 -> y ; S z -> S (z + y)}`.

Solución

(a)
```
dejar :: N -> [a] -> [a]
dejar = \n l -> case n of {0 -> [];
                           S x -> case l of {[] -> [] ;
                                             z:zs -> z : dejar x zs }}
```

(b)
```
sacar :: N -> [a] -> [a]
sacar = \n l -> case n of {0 -> l ;
                           S x -> case l of {[] -> []
                                             z:zs -> sacar x zs }}}
```

(c) `(∀n::N)(∀m::N)(∀l::[a]) sacar n (dejar (n+m) l) = dejar m (sacar n l)`

Dem. Por inducción en `n::N`

Caso 0: `(∀m::N)(∀l::[a]) sacar 0 (dejar (0+m) l)` $\stackrel{?}{=}$ `dejar m (sacar 0 l)`

Dem. Sean `m::N, l::[a]`.

`sacar 0 (dejar (0+m) l)`	`dejar m (sacar 0 l)`
= (def. `(+)`)	= (def. `sacar`)
`sacar 0 (dejar m l)`	`dejar m l`
= (def. `sacar`)	
`dejar m l`	

Ambas expresiones son iguales por reducción a la misma expresión.

Caso S: Sea `x::N`

HI) `(∀m::N)(∀l::[a]) sacar x (dejar (x+m) l) = dejar m (sacar x l)`

TI) `(∀m::N)(∀l::[a]) sacar (S x) (dejar ((S x)+m) l)` $\stackrel{?}{=}$ `dejar m (sacar (S x) l)`

Dem. Sea `m::N`. Por inducción en `l::[a]`.

Caso [ ]: `sacar (S x) (dejar ((S x)+m) [])` $\stackrel{?}{=}$ `dejar m (sacar (S x) [])`

`sacar (S x) (dejar ((S x)+m) [])`	`dejar m (sacar (S x) [])`
= (def. `(+)`)	= (def. `sacar`)
`sacar (S x) (dejar (S(x+m)) [])`	`dejar m []`
(def. `dejar`)	= (def. `dejar`)
`sacar (S x) []`	`[]`
= (def. `sacar`)	
`[]`	

Ambas expresiones son iguales por reducción a la misma expresión.

Caso (:): Sea `zs::[a]`.

HI2) `dejar m (sacar (S x) zs) = sacar (S x) (dejar ((S x)+m) zs)`

TI2) $(\forall$`z::a`$)$ `sacar (S x) (dejar ((S x)+m) (z:zs))` $\stackrel{?}{=}$ `dejar m (sacar (S x) (z:zs))`

Dem. Sea `z::a`.

`sacar (S x) (dejar ((S x)+m) (z:zs))`	`dejar m (sacar (S x) (z:zs))`
= (def. `(+)`)	= (def. `sacar`)
`sacar (S x) (dejar (S(x+m)) (z:zs))`	`dejar m (sacar x zs)`
= (def. `dejar`)	= (HI de arriba para `l = zs`)
`sacar (S x) (z:(dejar (x+m) zs))`	`sacar x (dejar (x+m) zs)`
= (def. `sacar`)	
`sacar x (dejar (x+m) zs)`	

Ambas expresiones son iguales por reducción a la misma expresión.

Nota: las funciones también podrían haberse definido haciendo casos primero en `l` y luego en `n` para el caso `(:)`. En ese caso, la demostración salía en forma directa haciendo primero inducción en la lista `l`, y en el caso `(:)` debía hacerse inducción en `n`.

Problema 2. [20p]

Considere la siguiente definición:

```
fun :: Eq a => N -> a -> T a -> Bool
fun = \m -> \y -> \z -> case z of { E e -> e /= y;
                                    H p r t -> case (p == y) of
                                               { False -> fun m y r;
                                                 True -> fun t p r };
                                    I n t j -> (fun m y n && fun 0 y t) || fun m y j;
                                    J p q a -> case q of
                                               { [] -> fun 0 y (E p);
                                                 e:es -> fun m a (E e) } }
```

(a) Defina el tipo `T a` para que la función `fun` compile:
`data T a where {E ::... ; H ::... ; I ::... ; J ::...}`

(b) ¿Cuáles constructores son semillas y cuáles son generadores? Justifique su respuesta.

(c) Defina la función `cantAs :: T a -> N` que dado un árbol `T a`, cuenta la cantidad de elementos de tipo a presentes en el mismo.
Puede utilizar las funciones del preludio de Haskell que considere necesarias.

Solución

(a)
```
data T a where { E :: a -> T a;
                 H :: a -> T a -> N -> T a;
                 I :: T a -> T a -> T a -> T a;
                 J :: a -> [a] -> a -> T a }
```

(b) `E` y `J` son semillas, ya que no utilizan elementos del tipo definido para construir nuevos elementos.
`H` e `I` son generadores, ya que utilizan elementos del tipo definido para construir nuevos elementos.

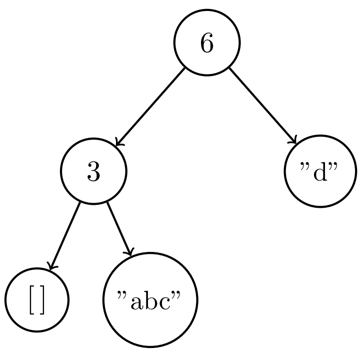
(c)
```
cantAs :: T a -> N
cantAs = \t -> case t of { E e -> S 0;
                           H p r t -> S (cantAs r);
                           I n t j -> cantAs n + cantAs t + cantAs j;
                           J p q a -> S (S (length q))}
```

Problema 3. [44p]

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

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

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



(b) Defina la función `cantb::T a b -> N` que recibe un árbol `t` y cuenta la cantidad de elementos de tipo `b` que tiene. Puede utilizar la suma (+) de naturales definida anteriormente.

(c) Defina la función `cantPa::T a b -> (a->Bool) -> N` que recibe un árbol `t` y un predicado `p` y cuenta la cantidad de elementos de tipo `a` en `t` para los cuales el predicado `p` devuelve `True`. Puede utilizar la suma (+) de naturales definida anteriormente.

(d) Defina la función `aplanar::T a b -> ([a],[b])` que recibe un árbol `t` y devuelve un par ordenado cuya primera componente contiene una lista con todos los elementos de tipo `a` de `t` y su segunda componente una lista con todos los elementos de tipo `b` de `t`. Puede utilizar la función `(++)` que concatena dos listas y las funciones `fst::(a,b)->a` y `snd::(a,b)->b` definidas en clase.

(e) Demuestre que (∀`t::T a b`)(∀`p::(a->Bool)` `cantPa t p` < `cantb t`
Puede utilizar las siguientes propiedades de <:

L1. (∀`x::N`) `0` < `S x`

L2. (∀`x,x',y,y'::N`) (`x<x'` ∧ `y<y'`) ⇒ `x+y` < `x'+y'`

L3. (∀`x,x',y,y'::N`) (`x<x'` ∧ `y<y'`) ⇒ `S(x+y)` < `x'+y'`

L4. Transitividad de <.

Solución

(a)
```
e :: T Int String
e = N (N (H[]) 3 (H''abc'')) 6 (H''d)
```

(b)
```
cantb :: T a b -> N
cantb = \t -> case t of { H x -> S 0;
                          N i e d -> cantb i + cantb d }
```

(c)
```
cantPa :: T a b -> (a -> Bool) -> N
cantPa = \t -> case t of { H x -> 0;
                           N i e d -> case p e of
                                      { False -> cantPa i p + cantPa d p;
                                        True -> S(cantPa i p + cantPa d p) }}
```

(d)
```
aplanar :: T a b -> ([a],[b])
aplanar = \t -> case t of { H x -> ([], [x]);
                            N i e d -> case aplanar i of {
                                       (ias, ibs) -> case aplanar d of {
                                          (das, dbs) -> (e : ias ++ das, ibs ++
dbs)}}}
```

(e) Demuestre que `(∀t::T a b)(∀p::(a->Bool))cantPa p t` $<$ `cantb t`

`Dem. Sea p::a->Bool.`

`Por inducción en t::T a b`

Caso H: `(∀x::b) cantPa p (H x)` $\overset{?}{<}$ `cantb (H x)`

`Dem. Sea x::b.`

`cantPa p (H x)`

`= (def. cantPa)`

`0`

$<$ `(L1)`

`S 0`

`= (def. cantb)`

`cantb (H x)`

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

HI1) `cantPa p i` $<$ `cantb i`

HI2) `cantPa p d` $<$ `cantb d`

TI) `(∀e::a) cantPa p (N i e d)` $\overset{?}{<}$ `cantb (N i e d)`

Dem. Sea `e::a`. Por casos en `p e::Bool`.

Caso p e = False:

`cantPa p (N i e d)`

= (def. `cantPa`)

`cantPa p i + cantPa p d`

$<$ (HI1, HI2, L2)

`cantb i + cantb d`

= (def. `cantb`)

`cantb (N i e d)`

Caso p e = True:

`cantPa p (N i e d)`

= (def. `cantPa`)

`S(cantPa p i + cantPa p d)`

$<$ (HI1, HI2, L3)

`S(cantb i + cantb d)`

= (def. `cantb`)

`cantb (N i e d)`.