PLP - Primer Parcial - 2do cuatrimestre de 2024

#Orden	Libreta	Apellido y Nombre	Ej1	Ej2	Ej3	Nota Final
[redacted]	[redacted]	[redacted]	E	E	E	A

(P) (para promoción!) ¡Excelente examen!

Este examen se aprueba obteniendo al menos dos ejercicios bien (B) y uno regular (R), y se promociona con al menos dos ejercicios muy bien (MB) y uno bien (B). Es posible obtener una aprobación condicional con un ejercicio muy bien (MB), uno bien (B) y uno insuficiente (I), pero habiendo entregado algo que contribuya a la solución del ejercicio. Las notas para cada ejercicio son: -, I, R, B, MB, E. Entregar cada ejercicio en hojas separadas. Poner nombre, apellido y número de orden en todas las hojas, y numerarlas. Se puede utilizar todo lo definido en las prácticas y todo lo que se dio en clase, colocando referencias claras. El orden de los ejercicios es arbitrario. Recomendamos leer el parcial completo antes de empezar a resolverlo.

Ejercicio 1 - Programación funcional

Aclaración: en este ejercicio no está permitido utilizar recursión explícita, a menos que se indique lo contrario.

El siguiente tipo de datos sirve para representar un buffer con historia que permite escribir o leer en cualquiera de sus posiciones (las posiciones tienen tipo `Int`). El tipo del buffer es paramétrico en el tipo de los contenidos que se pueden guardar en él. Si se escribe dos veces en la misma posición, el nuevo contenido pisa al anterior (por simplicidad). La lectura elimina el contenido leído.[1]

```
data Buffer a = Empty | Write Int a (Buffer a) | Read Int (Buffer a)
```

Definimos el siguiente buffer para los ejemplos:

```
buf = Write 1 ''a'' $ Write 2 ''b'' $ Write 1 ''c'' $ Empty
```

a) Dar el tipo y definir la función `foldBuffer` y `recBuffer`, que implementan respectivamente los esquemas de recursión estructural y primitiva para el tipo `Buffer a`. Solo en este inciso se permite usar recursión explícita.

b) Definir la función `posicionesOcupadas::Buffer a -> [Int]`, que lista las posiciones ocupadas en un buffer (sin posiciones repetidas).

Por ejemplo: `posicionesOcupadas buf` $\rightsquigarrow$ `[1, 2]`.

c) Definir la función `contenido::Int -> Buffer a -> Maybe a`, que devuelva el contenido de una posición en un buffer si hay algo en ella, y `Nothing` en caso contrario.

Por ejemplo:
`contenido 1 buf` $\rightsquigarrow$ `Just ''a''`.
`contenido (-2) buf` $\rightsquigarrow$ `Nothing`.
`contenido 1 (Read 1 buf)` $\rightsquigarrow$ `Nothing`.

d) Definir la función `puedeCompletarLecturas::Buffer a -> Bool`, que indique si todas las lecturas pueden completarse exitosamente (es decir, si cada vez que se intenta leer una posición, hay algo escrito en ella).

Por ejemplo:
`puedeCompletarLecturas (Read 1 Empty)` $\rightsquigarrow$ `False`.
`puedeCompletarLecturas (Read 1 buf)` $\rightsquigarrow$ `True`.
`puedeCompletarLecturas (Read 1 $ Read 1 buf)` $\rightsquigarrow$ `False`.

e) Definir la función `deshacer::Buffer a -> Int -> Buffer a`, que dados un buffer y un número natural n (es decir, un Int que por contexto de uso no es negativo), deshaga las últimas n operaciones del buffer, sacando los n constructores más externos para obtener un buffer como el original antes de realizar dichas operaciones. Si se realizaron menos de n operaciones, el resultado debe quedar vacío.

Por ejemplo: `deshacer 2 buf` $\rightsquigarrow$ `Write 1 ''c'' Empty`.

Pista: aprovechar la currificación y utilizar evaluación parcial.

[1] Por ejemplo, los buffers `b1 = Write 2 True Empty`, `b2 = Write 2 True (Write 2 False Empty)` y `b3 = Read 1 (Write 2 True (Write 1 False Empty))` tienen todos el valor `True` en la posición 2 y nada en las demás, aunque tienen distinta historia y podrían distinguirse mirando el historial de operaciones realizadas.

Ejercicio 2 - Demostración de propiedades

Considerar las siguientes definiciones[2]:

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

     const :: a -> b -> a                    altura :: AB a -> Int
{C}  const = (\ x -> \ y -> x)        {A0}   altura Nil = 0
                                     {A1}   altura (Bin i r d) = 1 + max (altura i) (altura d)

     zipAB :: AB a -> AB b -> AB (a,b)
{Z0} zipAB Nil = const Nil
{Z1} zipAB (Bin i r d) = \t -> case t of
                                 Nil -> Nil
                                 Bin i' r' d' -> Bin (zipAB i i') (r,r') (zipAB d d')
```

a) Demostrar la siguiente propiedad:

$$\forall\, \texttt{t::AB a} \,.\, \forall\, \texttt{u::AB a} \,.\, \texttt{altura t} \geq \texttt{altura (zipAB t u)}$$

Se recomienda hacer inducción en un árbol, utilizando extensionalidad para el otro cuando sea necesario. Se permite definir macros (i.e., poner nombres a expresiones largas para no tener que repetirlas).

No es obligatorio escribir los $\forall$ correspondientes en cada paso, pero es importante recordar que están presentes. Recordar también que los $=$ de las definiciones pueden leerse en ambos sentidos.

Se consideran demostradas todas las propiedades conocidas sobre enteros y booleanos, así como también:

$$\{\texttt{LEMA}\}\ \forall \texttt{t::AB a.altura}\ t \geq 0$$

b) Demostrar el siguiente teorema usando deducción natural, sin utilizar principios clásicos:

$$((\rho \wedge \sigma) \vee (\rho \wedge \tau)) \Rightarrow (\sigma \wedge \rho) \vee \tau$$

Ejercicio 3 - Cálculo Lambda Tipado

Se desea extender el cálculo lambda simplemente tipado para modelar **árboles ternarios**. Para eso se extienden los tipos y expresiones de la siguiente manera:

$\tau ::= \cdots \mid \mathtt{AT}(\tau)$

$M ::= \cdots \mid \mathtt{TNil}_\tau \mid \mathtt{Tern}(M, M, M, M) \mid \mathsf{foldAT}\ M \triangleright\ \mathsf{TNil} \rightsquigarrow M;\ \mathsf{Tern}(x, ri, rm, rd) \rightsquigarrow M$

- $\mathtt{AT}(\tau)$ es el tipo de los árboles ternarios con nodos de tipo τ.
- $\mathtt{TNil}_\tau$ es un árbol ternario vacío que admite valores de tipo τ.
- $\mathtt{Tern}(M_1, M_2, M_3, M_4)$ es un árbol ternario con raíz M_1 e hijos M_2, M_3 y M_4.
- $\mathsf{foldAT}\ M_1 \triangleright\ \mathsf{TNil} \rightsquigarrow M_2;\ \mathsf{Tern}(x, ri, rm, rd) \rightsquigarrow M_3$ es el esquema de recursión estructural para árboles ternarios, donde las variables x, ri, rm y rd se ligarán a la raíz y los resultados de la recursión sobre los hijos izquierdo, medio y derecho del árbol M_1 respectivamente.

a) Introducir las reglas de tipado para la extensión propuesta.

b) Definir el conjunto de valores y las nuevas reglas de reducción en un paso.

c) Mostrar paso por paso cómo reduce la expresión:

$$(\lambda t{:}\,\mathtt{AT(Nat)}.\mathsf{foldAT}\ t \triangleright\ \mathsf{TNil} \rightsquigarrow \mathsf{False};\ \mathsf{Tern}(x, ri, rm, rd) \rightsquigarrow \mathsf{isZero}(\mathsf{Pred}(x)))\ \mathtt{Tern}(\underline{1}, \mathtt{TNil}_{\mathtt{Nat}}, \mathtt{TNil}_{\mathtt{Nat}}, \mathtt{TNil}_{\mathtt{Nat}})$$

d) Definir como macro la función $\mathtt{mapAT}_{\sigma,\tau}$, que toma un $\mathtt{AT}(\sigma)$ y una función de σ en τ, y aplica dicha función a todos los elementos del árbol.

[2] Escritas con recursión explícita para facilitar las demostraciones.