PLP - Primer Recuperatorio - 2^{do} cuatrimestre de 2024

#Orden	Libreta	Apellido y Nombre	Ej1	Ej2	Ej3	Nota Final

Este examen se aprueba obteniendo al menos dos ejercicios bien menos (`B-`) y uno regular (`R`). Las notas para cada ejercicio son: `-`, `I`, `R`, `B-`, `B`. 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

En este ejercicio no se permite utilizar recursión explícita, a menos que se indique lo contrario.

El siguiente tipo de datos sirve para representar *operadores* que realizan operaciones combinadas sobre números enteros. Por simplicidad, modelaremos solamente las sumas y divisiones enteras.

`data Operador = Sumar Int | DividirPor Int | Secuencia [Operador]`

`Sumar n` representa la operación que suma `n` a un número entero. `DividirPor n` representa la operación que divide a un entero por `n` (descartando el resto). `Secuencia ops` representa la composición a izquierda de todas las operaciones en `ops`. En otras palabras, representa la operación de aplicar a un número todas las operaciones en `ops` de izquierda a derecha, siendo resultado de cada operación la entrada de la siguiente.

Por ejemplo: `Secuencia [Sumar 5, DividirPor 2]` representa la operación que, dado un entero, le suma 5, y al resultado lo divide por 2.

a) Dar el tipo y definir la función `foldOperador`, el esquema de recursión estructural para el tipo `Operador`. Sólo en este inciso se permite usar recursión explícita.

b) Definir la función `falla::Operador -> Bool`, que indica si un operador contiene una división por 0 entre sus operaciones. Por ejemplo:
`falla (Secuencia [Sumar 5, DividirPor 2])` ⇝ `False`.
`falla (Secuencia [Sumar 5, DividirPor 0])` ⇝ `True`.

c) Definir la función `aplanar :: Operador -> Operador`, que mapea un operador a uno equivalente pero de un solo nivel (si es una suma o división lo deja igual, si es una secuencia devuelve una secuencia con todas las sumas y divisiones en el orden en el que se realizan, sin agruparlas en subsecuencias). Por ejemplo:
`aplanar (Sumar 1)` ⇝ `Sumar 1`.
`aplanar (Secuencia [Sumar 1, Secuencia [DividirPor 3, Sumar 2]])` ⇝ `Secuencia [Sumar 1, DividirPor 3, Sumar 2]`.

Sugerencia: usar `concatMap` o una función auxiliar que arme una lista de sumas y divisiones.

d) Definir la función `componerTodas:: [a->a] -> (a->a)` que, dada una lista de funciones, devuelve el resultado de componerlas todas a izquierda (si la lista es vacía, devuelve la identidad).

Es decir: `componerTodas [f1,f2,...,fN] =((f1 . f2) . ...fN)`

e) Definir la función `aplicar::Operador -> Int -> Maybe Int`, que devuelve el resultado de aplicar el Operador a un número, o `Nothing` en caso de falla. Por ejemplo:
`aplicar (Secuencia [Sumar 5, DividirPor 2]) 2` ⇝ `Just 3`.
`aplicar (Secuencia [Sumar 5, DividirPor 0]) 2` ⇝ `Nothing`.

Pista 1: hacer una función auxiliar que aplique el operador a un entero suponiendo que no falla.

Pista 2: aprovechar la currificación y utilizar evaluación parcial.

Ejercicio 2 - Demostración de propiedades
Considerar las siguientes definiciones[1]:

```
    const :: a -> b -> a
{C} const = (\ x -> \ y -> x)

    head :: [a] -> a
{H} head (x:xs) = x

    tail :: [a] -> [a]
{T} tail (x:xs) = xs
```

```
     length :: [a] -> Int
{L0} length [] = 0
{L1} length (x:xs) = 1 + length xs

     null :: [a] -> Bool
{N0} null [] = True
{N1} null (x:xs) = False
```

```
     zip :: [a] -> [b] -> [(a,b)]
{Z0} zip []     = const []
{Z1} zip (x:xs) = \ys -> if null ys then [] else (x, head ys):zip xs (tail ys)
```

a) Demostrar la siguiente propiedad:

$$\forall\ \texttt{xs::[a]}\ .\ \forall\ \texttt{ys::[a]}\ .\ \text{length (zip xs ys)} = \text{min (length xs) (length ys)}$$

Se recomienda hacer inducción en una de las listas, utilizando extensionalidad para la otra 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:

$$\{\texttt{LEMA}\}\ \forall\texttt{xs::[a]}\texttt{.min (length}\ xs\texttt{)}\ 0 = 0$$

b) Demostrar el siguiente teorema usando deducción natural: $(\tau \Rightarrow (\sigma \wedge \rho)) \vee (\rho \Rightarrow (\sigma \Rightarrow \tau))$.

Se permite utilizar **principios clásicos**. (**Sugerencia:** usar $\tau \vee \neg\tau$).

Ejercicio 3 - Cálculo Lambda Tipado
Se desea extender el Cálculo Lambda tipado con colas bidireccionales (también conocidas como *deque*). No vamos a definir aquí los observadores (ya que están en la guía), sino que vamos a enfocarnos en los constructores y el esquema de recursión primitiva.
Se extenderán los tipos y términos de la siguiente manera:
$\tau ::= \cdots \mid \mathsf{Cola}_\tau \qquad M ::= \cdots \mid \langle\rangle_\tau \mid M \bullet M \mid \mathsf{recr}\ M \triangleright \langle\rangle \rightsquigarrow M; r, c \bullet x \rightsquigarrow M$
donde $\langle\rangle_\tau$ es la cola vacía en la que se pueden encolar elementos de tipo τ; $M_1 \bullet M_2$ representa el agregado del elemento M_2 al **final** de la cola M_1; y el esquema de recursión primitiva $\mathsf{recr}\ M_1 \triangleright \langle\rangle \rightsquigarrow M_2; r, c \bullet x \rightsquigarrow M_3$ permite operar con la cola en sentido contrario, accediendo al último elemento encolado (cuyo valor se ligará a la variable x en M_3), al resto de la cola (que se ligará a la variable c en el mismo subtérmino) y al resultado de la recursión sobre el resto de la cola (que se ligará a la variable r).

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

b) Demostrar la validez del siguiente juicio de tipado:

$\emptyset \vdash \lambda c\colon \mathsf{Cola}_{\mathsf{Nat}}.\mathsf{recr}\ c \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{\mathsf{Bool}}; r, y \bullet x \rightsquigarrow r \bullet \mathsf{True}\colon \mathsf{Cola}_{\mathsf{Nat}} \to \mathsf{Cola}_{\mathsf{Bool}}$

c) Definir el conjunto de valores y las nuevas reglas de semántica.

d) Mostrar paso por paso cómo reduce la expresión (pueden definir macros para no escribir muchas veces lo mismo.):

$\mathsf{recr}\ \langle\rangle_{\mathsf{Nat}} \bullet \mathsf{zero} \bullet \underline{1} \triangleright \langle\rangle \rightsquigarrow \langle\rangle_{\mathsf{Nat}}; r, c \bullet x \rightsquigarrow \mathsf{if\ isZero}(x)\ \mathsf{then}\ c\ \mathsf{else}\ r \bullet x$

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