PLP - Primer Parcial - 1^{er} cuatrimestre de 2024

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

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 árboles ternarios:

```
data AT a = NilT | Tri a (AT a) (AT a) (AT a)
```

Definimos el siguiente árbol para los ejemplos:

```
at1 = Tri 1 (Tri 2 NilT NilT NilT) (Tri 3 (Tri 4 NilT NilT NilT) NilT NilT)
      (Tri 5 NilT NilT NilT)
```

a) Dar el tipo y definir la función `foldAT` que implementa el esquema de recursión estructural para el tipo `AT a`. Sólo en este inciso se permite usar recursión explícita.

b) Definir la función `preorder :: AT a -> [a]`, que lista los nodos de un árbol ternario en el orden en que aparecen: primero la raíz, después los nodos del subárbol izquierdo, luego los del medio y finalmente los del derecho.

   Por ejemplo: `preorder at1` $\rightsquigarrow$ `[1, 2, 3, 4, 5]`.

c) Definir la función `mapAT :: (a -> b) -> AT a -> AT b`, análoga a la función map para listas, pero para árboles ternarios.

   Por ejemplo: `mapAT (+1) at1` $\rightsquigarrow$ `Tri 2 (Tri 3 NilT NilT NilT) (Tri 4 (Tri 5 NilT NilT) NilT NilT) (Tri 6 NilT NilT NilT)`.

d) Definir la función `nivel :: AT a -> Int -> [a]`, que devuelve la lista de nodos del nivel correspondiente del árbol, siendo 0 el nivel de la raíz.

   Por ejemplo: `nivel at1 1` $\rightsquigarrow$ `[2, 3, 5]`.

   Pista: aprovechar la currificación y utilizar evaluación parcial.

Ejercicio 2 - Demostración de propiedades

Considerar las siguientes definiciones sobre listas y árboles estrictamente binarios[1]:

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)

    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
```

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

```
     altura :: AEB a -> Int
{A0} altura (Hoja x) = 1
{A1} altura (Bin i r d) = 1 + max (altura i) (altura d)

     esPreRama :: Eq a => AEB a -> [a] -> Bool
{E0} esPreRama (Hoja x) = \xs -> null xs || (xs == [x])
{E1} esPreRama (Bin i r d) = \xs -> null xs ||
                    (r == head xs && (esPreRama i (tail xs) || esPreRama d (tail xs)))
```

a) Asumiendo `Eq a`, demostrar la siguiente propiedad:

   $\forall$ `t::AEB a` . $\forall$ `xs::[a]` . esPreRama t xs $\Rightarrow$ length xs $\leq$ altura t

   Se recomienda hacer inducción en el árbol, utilizando extensionalidad de booleanos y listas 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 que $\forall$ `t::AEB a` . altura t ≥ 0.

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

   $((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow \neg R \Rightarrow \neg P$

Ejercicio 3 - Cálculo Lambda Tipado

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

$\tau ::= \cdots \mid \texttt{Dicc}(\tau, \tau)$

$M ::= \cdots \mid \texttt{Vacío}_{\sigma,\tau} \mid \texttt{definir}(M, M, M) \mid \texttt{def?}(M, M) \mid \texttt{obtener}(M, M)$

- $\texttt{Dicc}(\sigma, \tau)$ es el tipo de los diccionarios con claves de tipo σ y valores de tipo τ.
- $\texttt{Vacío}_{\sigma,\tau}$ es un diccionario vacío con claves de tipo σ y valores de tipo τ.
- $\texttt{definir}(M, N, O)$ define el valor O en el diccionario M para la clave N.
- $\texttt{def?}(M, N)$ indica si la clave N fue definida en el diccionario M.
- $\texttt{obtener}(M, N)$ da el valor asociado a la clave N en el diccionario M (se espera que el diccionario tenga definida la clave y, en caso contrario, la expresión puede tipar, pero no se obtendrá un valor).

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

b. Definir el conjunto de valores y las nuevas reglas de semántica operacional a pequeños pasos. Suponer que el tipo de las claves cuenta con el operador `==` (es decir, el cálculo está extendido con un operador de comparación para los tipos que se usen como claves). Para el caso de $\texttt{obtener}(M, N)$, se espera que la clave N esté definida en el diccionario M. Si no lo está, puede colgarse o terminar en una expresión de error (una forma normal que no es un valor). **No es necesario escribir las reglas de congruencia**, sino que basta con indicar cuántas son.

c. Mostrar paso por paso cómo reduce la expresión:

   $(\lambda d\colon \texttt{Dicc}(\mathsf{Nat}, \mathsf{Bool}).\mathsf{if}\ \texttt{def?}(d, \underline{0})\ \mathsf{then}\ \texttt{obtener}(d, \underline{0})\ \mathsf{else}\ \mathsf{False})\ \texttt{definir}(\texttt{Vacío}_{\mathsf{Nat},\mathsf{Bool}}, \underline{0}, \mathsf{True})$

   Suponer que $\mathsf{zero} == \mathsf{zero} \rightarrow \mathsf{True}$.