PLP - Primer Parcial - 1er cuatrimestre de 2015

Este examen se aprueba obteniendo al menos **65 puntos** en total, y al menos **5 puntos** por cada tema. Poner nombre, apellido y número de orden en cada hoja, y numerarlas. Se puede utilizar todo lo definido en las prácticas y todo lo que se dio en clase, colocando referencias claras.

Ejercicio 1 - Programación funcional (35 puntos)

Durante este ejercicio **no** se puede usar recursión explícita, a menos que se indique lo contrario. Para resolver un ítem pueden utilizarse las funciones definidas en los items anteriores, más allá de si fueron resueltos correctamente o no.

a. Escribir la función `entrelazar::[a]->[a]->[a]`, que dadas dos listas `xs` e `ys` devuelve una lista con los elementos de `xs` e `ys` intercalados. Por ejemplo:

   `entrelazar [1,1,1] [2,2,2,2]` $\rightsquigarrow$ `[1,2,1,2,1,2,2]`

   `entrelazar [1,1,1,1] [2,2,2]` $\rightsquigarrow$ `[1,2,1,2,1,2,1]`

   Debe funcionar también cuando una o ambas listas son infinitas.

b. Escribir la función `duplicarApariciones::[a]->[a]`, que dada una lista `xs` devuelve otra lista con los elementos de `xs` repetidos en posiciones contiguas. Por ejemplo:

   `duplicarApariciones [1,2,3,4,5]` $\rightsquigarrow$ `[1,1,2,2,3,3,4,4,5,5]`

El siguiente es un esquema de que representa la **programación dinámica** sobre naturales.

```
dynprog::([a]->a)->a->Int->[a]
dynprog _ x 0 = [x]
dynprog f x n | n > 0 = let rec = dynprog f x (n-1) in (f rec):rec
```

El parámetro `f` es una función que toma la lista de resultados de los pasos `0` a `n-1` (**en orden inverso**) y devuelve el resultado del paso `n`. El parámetro `x` es el resultado para el caso `n = 0`. Notar que la lista de resultados precalculados que recibe `f` tiene **longitud** `n` y nunca es vacía, y que el resultado para el paso `i` está en la posición `n-1-i` (y por ende resultado para el paso `n-i` está en la posición `i-1`). Para obtener el resultado final para el paso `n`, basta con aplicar `head` al resultado de `dynprog f n`. Este esquema permite representar la recursión global.

Por ejemplo, la siguiente función representa la división entera por un divisor positivo.

```
dividirPor::Int->Int->Int
dividirPor d | d > 0 = head . (dynprog f 0) where
    f = \res->if length res < d then 0 else 1 + (res!!(d-1))
```

Recordar que `res!!(d-1)` es el resultado de `dividirPor d (n-d)`.

Otro ejemplo:

`dynprog (\xs->head xs+length xs) 0 5` $\rightsquigarrow$ `[15,10,6,3,1,0]`

En la posición i de la lista está la suma de los enteros de 0 a $5 - i$.

c. Utilizando el esquema `dynprog`, definir la función `factorial` (y dar su tipo).

d. Definir, utilizando `dynprog`, la función `fibonacci::Int->[Int]`, que dado un natural `n` devuelve la lista de los `n+1` primeros números de la sucesión de `fibonacci`.

   Ejemplo: `fibonacci 7` $\rightsquigarrow$ `[1,1,2,3,5,8,13,21]`

e. Reescribir la función `listasQueSuman::Int->[[Int]]` (vista en clase) en términos de `dynprog`. Esta función, dado un número natural `n`, devuelve todas las listas de enteros positivos[1] cuya suma sea `n`.

   Ejemplo: `listasQueSuman 4` $\rightsquigarrow$ `[[1,1,1,1],[1,1,2],[1,2,1],[1,3],[2,1,1],[2,2],[3,1],[4]]`

[1] Es decir, mayores o iguales que 1

Ejercicio 2 - Cálculo Lambda Tipado (35 puntos)
Se desea extender el cálculo lambda tipado para tener un mayor control sobre el proceso de reducción. Para esto, se introducen expresiones capaces de detener la reducción de un término, o de continuar una reducción que estaba detenida.

El conjunto de tipos será: $\sigma ::= ... \mid \texttt{det}(\sigma)$ donde $\texttt{det}(\sigma)$ es el tipo de los términos que resultan de detener la reducción de términos de tipo σ.
El conjunto de términos será: $M ::= ... \mid \texttt{detener}(M) \mid \texttt{continuar}(M)$
El comportamiento de estas expresiones es el siguiente: sea M un término tipable cualquiera, $\texttt{detener}(M)$ detiene la reducción de M. Es decir, no reduce por más que M pueda reducirse. Por otro lado, si N es un término detenido, $\texttt{continuar}(N)$ reanuda la reducción de N.
Por ejemplo, $\texttt{continuar}((\lambda x\colon \texttt{det}(\mathsf{Nat}).x)\ \texttt{detener}(\mathsf{Pred}(\mathsf{Succ}(0)))) \to \texttt{continuar}(\texttt{detener}(\mathsf{Pred}(\mathsf{Succ}(0)))) \to \mathsf{Pred}(\mathsf{Succ}(0)) \to 0$.
Además, las funciones que esperan argumentos detenidos, pueden recibir argumentos del tipo correspondiente sin detener. En ese caso, en lugar de reducir el argumento hasta obtener un valor, lo detienen. Esto permite definir funciones que toman parámetros por nombre (call-by-name). Por ejemplo:
$(\lambda x\colon \texttt{det}(\mathsf{Nat}).\mathsf{if\ True\ then}\ \texttt{continuar}(x)\ \mathsf{else}\ 0)\ \mathsf{Succ}(\mathsf{Pred}(\mathsf{Succ}(0))) \to$
$(\lambda x\colon \texttt{det}(\mathsf{Nat}).\mathsf{if\ True\ then}\ \texttt{continuar}(x)\ \mathsf{else}\ 0)\ \texttt{detener}(\mathsf{Succ}(\mathsf{Pred}(\mathsf{Succ}(0)))) \to$
$\mathsf{if\ True\ then}\ \texttt{continuar}(\texttt{detener}(\mathsf{Succ}(\mathsf{Pred}(\mathsf{Succ}(0))))) \mathsf{\ else\ } 0 \to$
$\texttt{continuar}(\texttt{detener}(\mathsf{Succ}(\mathsf{Pred}(\mathsf{Succ}(0))))) \to \mathsf{Succ}(\mathsf{Pred}(\mathsf{Succ}(0))) \to \mathsf{Succ}(0)$.

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

b. Exhibir la derivación de tipado para el siguiente juicio:
$\{y\colon \mathsf{Bool}\} \triangleright (\lambda x\colon \texttt{det}(\mathsf{Bool}).\mathsf{if}\ y\ \mathsf{then}\ \texttt{continuar}(x)\ \mathsf{else\ False})\ \mathsf{isZero}(0)\colon \mathsf{Bool}$.

c. Indicar formalmente cómo se modifica el conjunto de valores, y dar la semántica operacional de a un paso para la extensión propuesta. Notar que puede ser necesario modificar alguna de las reglas preexistentes.

Ejercicio 3 - Inferencia (30 puntos)
Se extendió el cálculo lambda tipado agregando unión de funciones. Para ello, extendimos el conjunto de términos y el de tipos de la siguiente manera:

$$M_1 \dots M_k ::= \cdots \mid [(M_1, \dots, M_k)] \qquad \sigma ::= \cdots \mid \mathsf{Union}(\sigma_1, \dots, \sigma_k)_\tau$$

Cada M_i dentro de "[()]" es una función con distinto dominio del resto pero con la misma imagen.
En el tipo $\mathsf{Union}(\sigma_1, \dots, \sigma_k)_\tau$, cada σ_i representa el tipo del dominio de M_i (la función en la posición i), y τ el tipo de la imagen de todas las funciones.

Al aplicarse esta unión sobre un valor de tipo σ, el término reduce utilizando la función de esta unión cuyo tipo para el dominio sea σ. Es decir, aplicando la función que corresponda según el dominio.

Incorporamos además las siguientes reglas de tipado:

$$\frac{\Gamma \triangleright M_1\colon \sigma_1 \to \tau \quad \dots \quad \Gamma \triangleright M_k\colon \sigma_k \to \tau \quad \forall i, j\ (i \neq j \Rightarrow \sigma_i \neq \sigma_j)}{\Gamma \triangleright [(M_1, \dots, M_k)]\colon \mathsf{Union}(\sigma_1, \dots, \sigma_k)_\tau}\ \mathsf{T\text{-}Union}$$

$$\frac{\Gamma \triangleright M\colon \mathsf{Union}(\sigma_1, \dots, \sigma_k)_\tau \quad \Gamma \triangleright N\colon \sigma_i \quad 1 \leq i \leq k}{\Gamma \triangleright MN\colon \tau}\ \mathsf{T\text{-}UApp}$$

a. Extender el algoritmo de inferencia para soportar la extensión propuesta.

b. Utilizar el algoritmo extendido para tipar $[((\lambda x.\mathsf{isZero}(x)), (\lambda y.y))]\mathsf{True}$, o demostrar que no tipa.