# Práctico 6 de Programación Funcional

UdelaR/FCien/CMat

xx al yy de octubre de 2016

1. Un *dominio de Scott* es un dcpo ($\mathcal{D} = (D, \sqsubseteq)$) *algebraico* y donde todo conjunto *coherente* de $D$ tiene supremo. Esto significa:

   - $\mathcal{D}$ es un poset, i.e.: un conjunto parcialmente ordenado.
   - Es un dcpo, i.e.: un poset donde todo conjunto dirigido $U \subseteq D$ tiene supremo.
   - Es algebraico, i.e.: un poset donde $\forall d \in D \quad \downarrow d$ es dirigido y $\sup(\downarrow d) = d$.
   - Un poset donde $\forall X \subseteq D$, si $X$ acotado superiormente (i.e.: $X$ es *coherente*), entonces $X$ tiene supremo.

   (a) Dar ejemplos de: i) Un poset que no sea dcpo, ii) Un dcpo que no sea algebraico, iii) un dcpo algebraico que no sea un dominio de Scott.

   (b) Demostrar que en un poset $\mathcal{D} = (D, \sqsubseteq)$, una función $f : D \to D$ que cumple $f(\sup X) = \sup f(X)$ para todo $X \subseteq D$ donde ambos supremos existen (en ese caso se dice que $f$ *conmuta con los supremos*), es creciente. Dar un ejemplo de un poset $\mathcal{D}$ y una función creciente $f : D \to D$ que no tenga un punto fijo. ¿Qué se puede afirmar sobre el poset $\mathcal{D}$?

   (c) Probar que si $\mathcal{D}$ y $\mathcal{E}$ son posets, $f : D \to E$ y $D$ es finito, entonces $f$ es creciente si y sólo si conmuta con los supremos.

2. Un dominio *flat* es un conjunto $X \cup \{\bot\}$ con la relación de orden $\sqsubseteq$:

   - $x \sqsubseteq y \iff x = y$ para todos los $x, y \in X$
   - $\bot \sqsubseteq x$ para todo $x \in X \cup \{\bot\}$.

   Los tipos primitivos de Haskell `Int, Integer, Float, Double, Bool, Char` (esto es, los enumerados) son dominios flat. Probar que los dominios flat son dominios de Scott. Probar que toda función estricta entre dominios flat es Scott-continua. ¿Es toda función Scott-continua computable? Justificar.

   Probar que una función no estricta a valores en un dominio flat es Scott-continua si y sólo si es constante.

3. Probar que si `a` y `b` son dominios de Scott, entonces `(a,b)` y `Either a b` son dominios de Scott.

   Probar que el constructor `Pair::a->b->(a,b)` y los destructores `fst::(a,b)->a` y `snd::(a,b)->b` son Scott-continuos.

   Probar un resultado análogo para los constructores y destructores de los tipos `Either a b` y `List a`.

4. Sean `a` y `b` dominios de Scott. Probar que el objeto exponencial `[a->b]` es un dominio de Scott.

5. Se consideran las funciones

```
curry            ::  ((a,b)->c)->a->b->c
curry f x y       =  f(x,y)
uncurry          ::  (a->b->c)->(a,b)->c
uncurry f (x,y)   =  f x y
```

   Probar que `curry` y `uncurry` son Scott-continuas.

6. Las conectivas booleanas binarias predefinidas en Haskell son *lazy* en el segundo argumento y se definen de la siguiente forma:

```
(&&)         ::  Bool->Bool->Bool
True && x    =   x
False && x   =   False
(||)         ::  Bool->Bool->Bool
True || x    =   True
False || x   =   x
```

   La negación se define del modo usual:

```
not         ::  Bool->Bool
not True    =   False
not False   =   True
```

   Probar que `&&, ||` y `not` son Scott-continuas. Probar que la igualdad `(==)::Bool->Bool->Bool` es Scott-continua.

7. Se consideran `cond::a->Bool`, `f1, f2::a->b` Scott-continuas. Probar que `g::a->b` definida como `g x=if (cond x) then (f1 x) else (f2 x)` es Scott-continua. Justificar cuidadosamente el resultado.

8. Se consideran `f1::b` y `f2::[a]->b` Scott-continua. Probar que `g::[a]->b` definida por *pattern-matching* como:

```
g []      =  f1
g (x:xs)  =  f2 (x:xs)
```

   es Scott-continua.

   Generalizar el resultado con parámetros, esto es cuando `f1::p1->...->pk->b` y `f2::p1->...pk->[a]->b` definen por pattern-matching en (el argumento de tipo) `[a]` una función `g::p1->...->pk->[a]->b`.

9. Probar que `lengthNat::[a]->Nat` definida como `lengthNat [] = Zero` y `lengthNat (x:xs) = Succ(lengthNat xs)` y la concatenación `(++)` en listas son Scott-continuas. (**Sug.:** usar el operador de punto fijo y el ejercicio anterior).

10. Sea `f::Integer->Integer`. Se define para cada natural $n$ la aproximación $n$-ésima de $f$, que definimos `approxf::Integer->Integer->Integer` según:

    - `approxf 0 x` = $\bot$
    - `approxf n x = if ((-n <= x)&&(x<=n))then (f x) else` $\bot$

    Probar que si `f` es Scott-continua, entonces `approxf` es Scott-continua para cada `n::Integer`. Probar que $\{$`approxf` $\ulcorner n \urcorner \mid n \in \mathbb{N}\}$ es una cadena cuyo supremo es `f`.

11. Se considera `Halt::(Integer->Integer)->Integer->Bool` definida como `Halt (f,x)=True` si `f` está definida en `x` y `Halt (f,x)=False` si no. Probar que esta función no es Scott-continua. Concluir que no es Haskell-computable.

12. Las funciones computables en Haskell son codificables mediante enteros, codificando el código Haskell que la define (por ejemplo, mediante una condificación de Gödel). En general, para cada función Haskell computable existen diversos códigos Haskell que la definen (justificar esto mediante ejemplos). Asignamos a cada función `f::Integer->Integer` Haskell computable el menor código natural de una posible definición en Haskell de `f`. Denotamos este código como $\ulcorner$`f`$\urcorner \in \mathbb{N}$. Probar que la función ($\ulcorner\ \urcorner$)`::(Integer->Integer)->Integer` no es Haskell computable (**Sug.:** suponer por absurdo que es Haskell-computable y llegar a una contradicción con el resultado del ejercicio anterior).