Práctico 5 de Programación Funcional

UdelaR/FCien/CMat

octubre de 2016

1. Implementar el tipo de datos `Nat` y las funciones `plus`, `times` y `exp` vistas en el teórico (que corresponden resp. a suma, producto y exponenciación). Declarar `Nat` como instancia de `Eq, Ord` y `Show`.

   (a) Probar que `Zero` es neutro (bilateral) de `plus` y que `plus` es asociativa. Para simplificar la escritura, adoptaremos la *notación infija* para las operaciones binarias y la simbología usual en matemática: `m+n` denota `plus m n`, `m`×`n` denota `times m n` y $\texttt{m}^{\texttt{n}}$ denota `exp m n`.

   (b) Para cada natural $n \in \mathbb{N}$ se define $\ulcorner n \urcorner$ (el *nombre* de n) como: $\ulcorner 0 \urcorner :=$ `Zero` y $\ulcorner n+1 \urcorner :=$ `Succ` $\ulcorner n \urcorner$.

   Definir `name :: Integer -> Nat` que calcule la función $\ulcorner \_ \urcorner$.

   (c) Probar que $\forall$ `n::Nat` $\forall$`p::Nat` parcial `n+p = p`.

   Probar que $\forall n \in \mathbb{N} \forall$ `p::Nat` parcial `p+`$\ulcorner n \urcorner$`=`$\underbrace{\texttt{Succ} \ldots \texttt{Succ}}_{n}$ `p`.

   Concluir que todo parcial es de la forma $\perp$`+` $\ulcorner n \urcorner$ para algún $n \in \mathbb{N}$.

   (d) Concluir que `+` no es conmutativa en todo el tipo `Nat`. ¿Qué sucede restringiéndola a los miembros finitos de `Nat`?

   (e) Probar que `*` es distributiva frente a `+` (en todo `Nat`).

   (f) Probar que $\ulcorner 1 \urcorner$ es neutro (bilateral) de `*` y que `*` es asociativa.

   (g) Probar que $\forall n \in \mathbb{N} \forall$ `x::Nat x*(`$\perp$ `+` $\ulcorner n \urcorner$`) =` $\perp$ `+(x*`$\ulcorner n \urcorner$`)`.

   Probar que $\forall m, n \in \mathbb{N}[m \neq 0 \Rightarrow (\perp$ `+` $\ulcorner n \urcorner)$`*` $\ulcorner m \urcorner = \perp$ `+` $\ulcorner n \urcorner]$

   (h) Concluir que `*` no es conmutativa en todo el tipo `Nat`. ¿Qué sucede restringiéndola a los miembros finitos de `Nat`?

   (i) Investigar si `x` $^{\texttt{(y+z)}} = (\texttt{x}\ ^{\texttt{y}}) \times (\texttt{x}\ ^{\texttt{z}})$ para todo `x, y, z::Nat`.

2. (a) Definir por *pattern mathching* la resta `(-)` en `Nat`, de modo que para todo $n > 0$, `Zero -` $\ulcorner n \urcorner$ `=` $\perp$.

   Probar que $\forall$ `x::Nat` $\forall$ `y::Nat (x+y)-y = x`.

   (b) Definir agregando cláusulas a la definición de `(-)`, una versión total `(-')` tal que `m -' n = Zero` si `m < n`.

(c) Consideremos la proposición $\mathcal{P}(\mathtt{n}) \equiv \exists m \in \mathbb{N}\ \mathtt{n\ -'}\ \ulcorner m \urcorner = \bot$. Demostrar por inducción que $\mathcal{P}(\mathtt{n})$ se cumple para todos los números parciales `n`. Demostrar que `infinity -' m = infinity` para todo `m::Nat` finito. Concluir que $\mathcal{P}(\mathtt{infinity})$ no se satisface.

Este ejemplo muestra que una fórmula acerca de listas que contiene cuantificadores existenciales y se cumple para datos parciales, no necesariamente se cumple para datos infinitos.

3. Sean `e1, e2::b1 -> ... -> bk -> c -> a` funciones computables en la $k+1$-ésima variable. Las funciones computables son *monótonas y continuas*, esto es, fijados `x1::b1, ..., xk::bk`:

   - Si $\mathtt{y1} \sqsubseteq_{\mathtt{c}} \mathtt{y2}$ entonces $\mathtt{e1(x1,\ ...,\ xk,\ y1)} \sqsubseteq_{\mathtt{a}} \mathtt{e1(x1,\ ...,\ xk,\ y2)}$. Ídem para `e2`.
   - Sea $\mathcal{C}$ una cadena de valores de tipo `c` y con supremo s`::c`. Entonces $\mathtt{Sup}_{y \in \mathcal{C}}\{\mathtt{e1(x1,...,xk,y)}\} = \mathtt{e1(x1,...,xk,s)}$, ídem para `e2`.

   Sea $\mathcal{P}(y) \equiv \forall \mathtt{x1::b1} \ldots \forall \mathtt{xk::bk}\ \ \mathtt{e1(x1,...,xk,y)} \sqsubseteq_{\mathtt{a}} \mathtt{e2(x1,...,xk,y)}$.

   (a) Probar que para toda cadena $\mathcal{C}$ con supremo s:

   $$\text{si } \forall x \in \mathcal{C}\ \ \mathcal{P}(\mathtt{c}), \text{ entonces } \mathcal{P}(\mathtt{s}).$$

   (b) La igualdad en `a` viene dada por: `x = y` si y sólo si `x` $\sqsubseteq_a$ `y` e `y` $\sqsubseteq_a$ `x` (esto es: dos datos son iguales si tienen la misma información). Una fórmula ecuacional es de la forma:

   $$\mathcal{P}(y) \equiv \forall \mathtt{x1::b1} \ldots \forall \mathtt{xk::bk}\ \ \mathtt{e1(x1,...,xk,y) = e2(x1,...,xk,y)}.$$

   Usando la parte anterior, concluir que las fórmulas ecuacionales que son válidas para datos parciales, son válidas para datos infinitos (e.g.: en `infinity::Nat`, en listas infinitas, etc.)

4. Bosquejar el orden de Scott para los tipos `Bool, (Bool, Bool), [Bool]`. Decir cuáles de las siguientes especificaciones de funciones se pueden programar, haciéndolo en caso afirmativo y justificando la imposibilidad en caso contrario:

   - `f(True:`$\bot$`) = True` y `f(True:False:`$\bot$`)=False`
   - `f(True:`$\bot$`) = True` y `f([True,False])=False`
   - `f(True:`$\bot$`) = True` y `f([True,False])=True`
   - `f(True,False)=`$\bot$ y `f(False,True)=True:`$\bot$
   - Se define

     ```
     trueFalse ::  Integer   -> [Bool]
     trueFalse n = if odd n  then True:(trueFalse (n+1))
                             else False:(trueFalse (n+1))
     ```

     ¿Es posible definir `f::Nat->[Bool]` tal que

$\forall n \in \mathbb{N}$ `f (⊥ + ⌜n⌝) = trueFalse n`?

5. *Generalizaciones del principio de recursión.* Dados tipos `a` y `b`, definimos la notación $\texttt{a}^0\texttt{-> b} = \texttt{b}$ y $\texttt{a}^{k+1}\texttt{->b} = \texttt{a->(a}^k\texttt{->b)}$ para todo natural k.

   (a) El esquema de recursión estructural con parámetros es el siguiente principio de construcción: Dadas $b : \mathbb{N}^k \to \mathbb{N}$ y $h : \mathbb{N}^{k+1} \to \mathbb{N}$, b y h definen una única función $f : \mathbb{N}^{k+1} \to \mathbb{N}$ tal que $f(0, \vec{x}) = b(\vec{x})$ y $f(n+1, \vec{x}) = h(f(n, \vec{x}), \vec{x})$.
   Definir $\texttt{f::Nat}^{k+1}\texttt{->Nat}$ mediante una instancia de `foldNat h' b'` para una funciones `h'` y `b'` convenientes (que dependerán de `h` y `b`).

   (b) El esquema de *recursión primitiva* es el siguiente principio de construcción: Dadas $b : \mathbb{N}^k \to \mathbb{N}$ y $h : \mathbb{N}^{k+2} \to \mathbb{N}$, b y h definen una única función $f : \mathbb{N}^{k+1} \to \mathbb{N}$ que satisface:
   - $f(\vec{x}, 0) = b(\vec{x})$.
   - $f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y))$.

   Definir para cada aridad k una función $\texttt{primRec::(Nat}^k\texttt{->Nat)->(Nat}^{k+2}\texttt{->Nat)->Nat}^{k+1}\texttt{->Nat}$ que implemente la recursión primitiva. Escribir mediante `foldNat` una función `primRec'` tal que `primRec=snd.primRec'`. ¿Es posible fusionar `snd.primRec'` en una definición que sólo involucre `foldNat`? Justificar.

6. En $\mathbb{N}$ se tienen las siguientes igualdades:

$$\begin{aligned} m+n &= \underbrace{\mathrm{succ}(\ldots(\mathrm{succ}}_{n}(m))\ldots) \\ m \times n &= \underbrace{m+(\ldots(m}_{n}+0)\ldots) \\ m^n &= \underbrace{m \times (\ldots(m}_{n} \times 1)\ldots) \end{aligned}$$

   Si denotamos $m \uparrow^0 n = \mathrm{succ}(n)$, $m \uparrow^1 n = m+n$, $m \uparrow^2 n = m \times n$ y $m \uparrow^3 n = m^n$, tenemos entonces una sucesión finita de operaciones binarias $\uparrow^k$ que satisfacen

$$m \uparrow^{k+1} (n+1) = m \uparrow^k (a \uparrow^{k+1} n)$$

   Si agregamos las condiciones $m \uparrow^k 0 = 1$ para todo $k \geq 3$, tenemos la cláusula anterior define una sucesión infinita de operaciones.

   Definir una función `hyperexp::Nat->Nat->Nat->Nat` tal que $\texttt{hyperexp}\ulcorner k \urcorner$ compute $\uparrow^k$ para todo natural k.

   Demostrar que en los enteros finitos, $\texttt{hyperexp}\ulcorner k \urcorner$ con $k = 0, 1, 2$ y 3 coinciden respectivamente con `(\x-> Succ)`, `plus`, `times` y `exp`. ¿Qué sucede para enteros parciales? ¿Qué sucede en `infinity`?

7. Dos tipos de datos `a` y `b` son isomorfos si existen funciones calculabes:

   ```
   btoa ::  b -> a
   atob ::  a -> b
   ```

   tales que:

   `btoa . atob = idA`,

   `atob . btoa = idB`,

   siendo `idA` e `idB` las identidades en `a` y `b` respectivamente. Demostrar que existen isomorfismos entre:

   (a) `(a,b)` y `data Pair a b = Pair a b`

   (b) `Bool` y `Either Bottom Bottom`

   (c) `Maybe a` y `Either a Bottom`

   (d) `String` e `Integer`

   (e) `[Bottom]` y `Nat`

   (f) `(a,(b,c))` y `((a,b),c)`

   (g) `(a,Either b c)` y `Either (a,b) (a,c)`

   En todos los casos, implementar las dos funciones, (y demostrar que cumplen lo pedido).

8. Considerar nuevamente el tipo de datos `Nat`

   (a) Definir una función que capture la recursión estructural sobre los naturales (en la literatura, suele llamarse fold o catamorfismo):
   `foldNat :: (t -> t) -> t -> Nat -> t`

   (b) Definir las funciones del ejercicio 1 usando `foldNat`

   (c) Implementar las funciones

   `dup :: Nat -> Nat`,

   `sum1 :: Nat -> Nat`,

   `sum2 :: Nat -> Nat`

   que duplican, y suman una o dos unidades a un natural, respectivamente.

   (d) Demostrar por inducción estructural que:
   `dup . sum1 = sum2 . dup`

   (e) Demostrar por inducción estructural que:
   `foldNat Succ Zero = id`