FUNDAMENTOS DE COMPUTACIÓN
Repartido: Los Números Naturales - Parte 1

1. Tipos infinitos

Hasta el momento hemos visto un método para introducir tipos cuyos objetos son dados explícitamente, por enumeración (llamados tipos finitos o enumerados).

Nos preguntamos ahora cómo introducir tipos con una cantidad infinita de valores.
Es claro que no podemos utilizar el método de enumerar esos valores uno por uno, tal como hemos hecho con los tipos finitos precedentes, porque no terminaríamos nunca de definir el tipo deseado.

Entonces, ¿cuál es el camino?

Lo que necesitamos es un dispositivo que **pueda describirse de manera finita y que sea capaz de producir una lista infinita de los valores del tipo a introducir.**

Imaginemos que tenemos un dispositivo como ese, En cada momento, habrá producido una lista finita de los valores: v_1, v_2, ...v_n.
El acto de **alargar** esta lista con un nuevo valor v_{n+1} deberá realizarse mediante una operación sobre la lista existente y por lo tanto puede ser expresada utilizando una idea adecuada de **función**.

Investiguemos entonces qué clase de función será necesaria:

Para empezar: ¿cuál sería el parámetro de entrada de esta función ?

Podría ser la lista completa de valores ya generados pero, de hecho, alcanza con disponer de variantes más simples. Nos servirá cualquier función que acepte como entradas a un número finito de los valores de la lista ya generada, siempre y cuando ella genere con ellos un valor **distinto** de los anteriores.

Por ejemplo, supongamos que queremos generar un nuevo tipo llamado A, y la función en cuestión se llama G y acepta tres parámetros, que son valores del tipo que estamos generando.
Entonces G :: A → A → A → A.

Si escogemos tres de los valores de la lista ya generada, digamos v_1, v_2 y v_3, entonces
G v_1 v_2 v_3 será un nuevo valor del tipo A y lo podremos agregar a la lista.
Una vez hecho esto, el nuevo valor está disponible para ser utilizado en la generación de un valor subsiguiente.

En efecto, G v_1 v_2 v_3 puede, por ejemplo, ser utilizado otra vez (junto a otros dos valores existentes) como argumento de la propia función G, lo cual dará origen a otro nuevo valor de tipo A, necesariamente distinto de todos los anteriormente generados.
Una función que cumpla este rol es llamada un **generador** (del tipo A).

La condición esencial de un generador es que cuando se aplica a (combinaciones de) argumentos distintos genera valores distintos.

Es importante darse cuenta de que cada valor generado, por ejemplo G v_1 v_2 v_3, **no tiene otra representación más que esa misma**. O, en otras palabras, **el valor de esa expresión es exactamente ella misma**.

Es decir, los generadores no son funciones que operen sobre sus argumentos calculando un resultado del tipo A expresado en términos más básicos.
Por el contrario, ellas constituyen la forma de expresión básica o primitiva de esos valores.
En este sentido, un generador como G es análogo a las constantes False o True.
Nadie duda del carácter de las dos últimas: son expresiones que ya están evaluadas.
Lo mismo ocurre con un generador G.
La diferencia es solamente que este último admite parámetros (y es, por lo tanto, una función).

No es, por supuesto, obligatorio que un tipo infinito A cuente con un único generador.
Cualquier cantidad finita de generadores será aceptable para definir un tipo infinito, pudiéndose en cada paso elegir uno de los generadores para dar lugar al siguiente valor.

Pero, disponer de los generadores no es suficiente.
Porque cada generador sólo es capaz de operar sobre valores ya existentes, es decir cuando la lista de valores ya generados no está vacía.

Entonces... ¿cuál es el primer valor de la lista?

Para echar a andar la maquinaria es necesario disponer de ciertos valores iniciales, que llamaremos **semillas** del proceso de generación.
Una forma fácil de dar estas semillas es por enumeración (habiendo en tal caso, obviamente, una cantidad finita de ellas).

De este modo tenemos completada una caracterización de los tipos infinitos:
Un tipo (de tamaño) infinito A es dado por ciertos valores iniciales llamados semillas y un número finito de generadores.

En adelante, semillas y generadores serán llamados uniformemente **constructores**, que es la terminología empleada en la teoría de los lenguajes de programación. Las semillas se llaman **constructores iniciales** y los generadores se llaman **constructores recursivos.**

2. Los Números Naturales

Definimos al tipo **N** de los números naturales como el tipo infinito más sencillo posible. Es decir, uno que posee:

- una única semilla, y
- un único generador, con un único parámetro.

La semilla es llamada **0** (“cero”) y el generador es **S** (la función “siguiente” o “sucesor”). Las reglas que definen al tipo **N** son entonces:

$$\frac{}{N \text{ tipo}}$$

y sus valores constructores anteriormente mencionados:

$$\frac{}{0 :: N} \qquad \frac{e :: N}{S\ e :: N}$$

En Haskell esto se define del siguiente modo:

```
data N where { 0 :: N  ; S :: N → N }
```

Los **valores de N** forman naturalmente la secuencia (según su orden de generación):

0 , S 0 , S (S 0) , S (S (S 0)) , ... , S (.....(S (S 0))....) , ...

Esta secuencia comienza con el valor inicial **0** y cada nuevo valor se obtiene aplicando el generador **S** al último valor previamente generado.

¿Qué relación existe entre estos objetos y los números naturales que todos conocemos? Acostumbramos desplegar a los números naturales como la secuencia: 0, 1, 2, 3,.....n, n+1,... que comienza por 0 y donde cada nuevo elemento es obtenido sumando uno al anterior.

No tenemos disponibles los números naturales en la forma que acabamos de desplegar. El tipo N que acabamos de introducir toma para el valor inicial (cero), con igual notación y nombre, pero la operación que pasa al siguiente natural es para nosotros el constructor recursivo S.

Comparemos ambas notaciones en detalle:

- La notación tradicional usa “n+1” mientras que nosotros escribimos “S n”.

Ahora bien, si por “+” se entiende la operación de suma y por “1” la unidad (el segundo natural) entonces la notación tradicional no puede ser la forma de definir a los naturales.

Eso es claro, puesto que la notación estará presuponiendo que la suma y el número 1 están disponibles **antes** de introducir los números naturales.

Luego, lo que se denota con la notación “+1” debe ser una operación que simplemente **genera un nuevo número natural** a partir de uno dado, y por consiguiente será verdaderamente más “natural” darle un nombre diferente al equívoco +1 (puesto que no es una suma). Esto lo que hemos hecho al introducir el constructor S.

- En particular, nosotros suponemos que los naturales subyacen en la secuencia en cuestión en ninguna notación particular.
Específicamente, **no asumimos la notación decimal ni la binaria, ni ninguna otra**. Este comentario viene al caso porque el uso tradicional conduce frecuentemente a confundir a los números con su representación más tradicional, que es la decimal.

Lo que nosotros estamos haciendo es introducir los números naturales en una notación minimal, o sea, la unaria. La idea es, en principio, usar un único símbolo para denotar los números (al contrario de los diez símbolos empleados en la notación decimal o los dos de la binaria).
Con un único símbolo, por ejemplo I, el número tres se escribirá como I I I.
Este método tiene el defecto de no tener una forma de representar explícitamente al cero.
Así que, introducimos un símbolo adicional para denotar al cero.
El símbolo S es el equivalente del I recién considerado.
Con nuestra notación, formar los numerales y computar el sucesor de un número son operaciones triviales y lo mismo ocurre con la función predecesor, que consiste apenas en borrar la S delantera del número recibido.

Podríamos decir que la notación unaria capta la esencia *sin adornos* de los números naturales.
En cambio, la notación decimal está destinada a presentar cada número mediante una secuencia de dígitos con un largo más manejable y es por ello conveniente para introducir datos y presentar resultados.

3. Métodos de programación de funciones es sobre N

Análisis de casos

Nuestro propósito a partir de ahora es, por tanto, reconstruir la matemática de los números naturales a partir de la definición que acabamos de dar.

Comencemos por identificar métodos para definir (o sea, programar) función es que operen con números naturales.

Para esto necesitamos un operador case adaptado a los naturales. Veamos una primera versión de este operador:

$$\frac{\mathbf{e :: N} \qquad \mathbf{e1 :: t} \qquad \mathbf{e2 :: t}}{\mathbf{case\ e\ of\ \{\ 0 \rightarrow e1\ ;\ S\ x \rightarrow e2\ \}\ ::\ t}}\ .$$

Observar que en la segunda rama del case estamos considerando que el discriminante (e) resulte ser un sucesor, o sea, una expresión de la forma S x para algún natural x.

Las igualdades serían, en un principio, las siguientes para este case:

case 0 of { 0 → e1 ; S x → e2 } = e1
case (S e) of { 0 → e1 ; S x → e2 } = e2

Esta primera versión nos permitirá definir funciones como **positivo :: N → Bool**, que retorna True cuando un natural es mayor que cero:

positivo = λn→ case n of { 0 → False ; S x → True }

Hagamos algunas cuentas:

positivo 0 $\rightarrow_{\text{def. positivo}}$ (λn→ case n of { 0 → False ; S x → True }) 0
$\rightarrow_{\beta}$ case 0 of { 0 → False ; S x → True }
$\rightarrow_{\text{case}}$ False.

positivo (S(S 0)) $\rightarrow_{\text{def. positivo}}$ (λn→ case n of { 0 → False ; S x → True}) (S(S 0))
$\rightarrow_{\beta}$ case S(S 0)) of { 0 → False ; S x → True}
$\rightarrow_{\text{case}}$ True.

Pero, ¿qué pasa si queremos definir una función un poco más complicada, como **predecesor :: N → N**, que calcula el anterior a un número natural?.

Claramente la función predecesor deberá borrar la primera S del número recibido.
Para el caso 0 vamos a considerar que el predecesor de 0 es 0, ya que no tenemos números negativos. Entonces:

predecesor = λn→ case n of { 0 → 0 ; S x → x }

Si queremos computar el valor de predecesor (S(S 0)), hacemos las siguientes cuentas:

predecesor (S(S 0)) $\rightarrow_{\text{def. positivo}}$ (λn→ case n of { 0 → 0; S x → x }) (S(S 0))
$\rightarrow_{\beta}$ case S(S 0)) of { 0 → 0; S x → x }
$\rightarrow_{\text{case}}$ x

Bueno... algo anduvo mal, ya que según nuestras cuentas el predecesor de dos es igual a x! ¿Cuál fue el problema?

Mirando la definición de la función: **predecesor = λn→ case n of { 0 → 0 ; S x → x }**, es evidente que en el caso en que el argumento n es de la forma S x, lo que queremos devolver es el x que se obtiene al instanciar n con S x, y no la variable x como hicimos arriba.

Revisemos entonces las reglas de tipado e igualdad correspondientes a la expresión case y observemos dos cosas:

1) En la expresión e2 tenemos permitido (y es en general útil) utilizar a la variable x (como hemos hecho en la definición de la función predecesor).
Luego, para dejar explícito que x (de tipo N) puede aparecer en e2, la premisa e2 :: t debe reemplazarse por **e2 :: t [x :: N]**, quedando la regla correcta de la forma:

$$\frac{\textbf{e :: N} \qquad \textbf{e1 :: t} \qquad \textbf{e2 :: t [x::N]}}{\textbf{case e of \{ 0 → e1 ; S x → e2 \} :: t}}$$

2) El resultado de la expresión case cuando el discriminante es de la forma **S e** será entonces e2, pero **instanciando la variable x** que aparece en ella por la expresión **e**.
Esta instanciación no es otra que la operación de sustitución que ya estudiamos, quedando la segunda igualdad del case de la forma:

case (S e) of { 0 → e1 ; S x → e2 } = e2[x:=e].

Notar que la variable x es local a la rama, o sea está ligada en la subexpresión S x → x. Esto implica que:
- El nombre x no tiene ninguna relevancia al exterior de la rama.
- Un case cuya segunda rama use cualquier otra variable z en lugar de x es equivalente siempre que se sustituya la variable x por z en e2 (es un caso de igualdad α).

Ahora sí, volvamos a computar predecesor (S(S 0)):

predecesor (S(S 0)) $\rightarrow_{\text{def. positivo}}$ (λn→ case n of { 0 → 0; S x → x }) (S(S 0))
$\rightarrow_{\beta}$ case S(S 0)) of { 0 → 0; S x → x }
$\rightarrow_{\text{case}}$ x [x := S 0]
= S 0.

¡y así obtenemos el resultado correcto!