FUNDAMENTOS DE COMPUTACIÓN
Repartido: Los Números Naturales - Parte 2

4. Recursión Primitiva

Los números naturales constituyen una secuencia infinita de objetos, que empieza en un objeto inicial (el 0) y donde cada objeto se obtiene aplicando el constructor S al anterior: **0, S 0, S(S 0), S(S(S 0)), . . .**

Del mismo modo, podemos pensar a cada función f definida en los naturales como una secuencia infinita que contiene los términos **f 0, f (S 0), f (S(S 0)), f (S(S(S 0))), . . .**

Nos interesará definir secuencias en las que cada término pueda definirse aplicando una cierta transformación al término precedente. Y una condición suficiente para que un mecanismo de esta clase funcione, es simplemente comenzar con un primer término, que se definirá directamente.

Esta es una idea simple, que nos proporciona el siguiente método para definir secuencias (o sea funciones) sobre N:

1. Proveer el **término inicial**
2. Proveer la **transformación** que, aplicada a un término, **produce el siguiente** término de la secuencia.

Supongamos entonces que queremos definir **f :: N → t** para un tipo t cualquiera.
Ahora observemos, para empezar, que:
- El término inicial de la secuencia no es otra cosa que el valor de la función en **0**. Llamemoslo **a**.
- Cada término subsiguiente será el resultado de f en un natural **sucesor** (o sea, en S x para cada x).
- El término (o resultado de **f**) en **S x** será escrito como una cierta expresión, llamémosla **r**.

Estas observaciones nos permiten comenzar a escribir la definición de f como sigue:

f = λn→ case n of { 0 → a; S x → r }

Ahora bien, hemos dicho también que **cada término subsiguiente al inicial es calculado a partir de su inmediato predecesor**.
En otras palabras, el resultado de f en cada **S x** se obtiene aplicando una transformación a f x. Esto deberá reflejarse en la expresión **r**, que deberá por lo tanto mencionar, en general, a **f x**.
Este hecho lo expresamos con la notación **r [f x]**.
Arribamos así a una “definición completa" de la función **f**:

f = λn→ case n of { 0 → a; S x → r [f x] }

Se trata, como es evidente, de otra definición esquemática. Los elementos del metalenguaje son varios:
- El nombre mismo de la función **f**, que se escogerá libremente en cada caso.
- El término inicial **a**. Lo llamaremos el resultado base, o simplemente la base de la definición.
- La expresión **r** que, construida a partir de la expresión **f x**, constituye el resultado **f (S x)**. La llamaremos el paso (recursivo) de la definición.

Esta definición esquemática es llamada el **Esquema de recursión primitiva en N** y provee un método muy general de programación de funciones sobre números naturales.
El método consiste solamente en completar adecuadamente la plantilla precedente, es decir:

1. Escribir la base **a**, que es el resultado de **f** en **0**.
2. Escribir el paso recursivo **r**, es decir el resultado de **f** en **S x** usando **f x**.

Vale la pena enfatizar este segundo ítem:
Para describir el resultado de la función f para el caso S x, podemos asumir <u>dado, disponible y correcto</u> el resultado de f x.
La razón es que el método sólo requiere que se indique como calcular cada término subsiguiente de la secuencia a partir de su inmediato predecesor.

Por si hace falta, es posible remarcar que, efectivamente, este método garantiza que la función **f** queda definida para todo natural. Porque claramente en **0** está definida (con resultado **a**). Pero, estando definida en **0**, el paso **r** garantiza que lo está también en **S 0**. Y por el mismo argumento, es decir utilizando nuevamente el paso **r**, lo está en **S (S 0))**, y así sucesivamente.
Queda claro entonces que la función **f** puede calcularse en cualquier número natural, comenzando por el resultado en **0** y aplicando sucesivamente el paso **r** hasta llegar al número requerido.

Un primer ejemplo sencillo es la función **par :: N → Bool** que devuelve **True** cuando el argumento es un número par.

Escribimos la expresión:

par = λn→ case n of { 0 → ?1 ; S x → ?2 }

donde las incógnitas deben rellenarse adecuadamente:

- ?1 debe ser el resultado de la función **par** en **0**. Como **0** es par, el resultado será **True.**

- Para escribir **?2**, comenzamos por asumir que **par x** está disponible (o sea, **sabemos cuánto vale par x**) y a partir de esta información **debemos escribir la expresión correspondiente a par (S x).**

¿Cómo podemos obtener el valor de **par (S x)** a partir de **par x**?
En este caso la respuesta es simple, puesto que **par (S x)** es simplemente el opuesto de **par x**, se obtiene negando este.

La solución final es, entonces:

par = λn→ case n of { 0 → True ; S x → not (par x) }

Observemos que en la definición anterior, la constante definida (**par**) aparece también a la derecha del signo = de la definición, o sea, donde supuestamente se está estableciendo su significado.

Las definiciones con esta particularidad se llaman recursivas, y la expresión **par x** que forma parte del paso recursivo es denominada la **llamada recursiva a la función**.

No toda definición recursiva tiene sentido. Por ejemplo, si en el caso **S x** hubiéramos escrito **not (par (S x))**, estaríamos ante una función que no termina, ya que, por ejemplo:

par (S 0) $\rightarrow_{\text{def. par, } \beta, \text{ case}}$ not (par (S 0)) $\rightarrow_{\text{def. par, } \beta, \text{ case}}$ not (not (par (S 0))) → .....

En la **recursión primitiva**, la recursión está limitada.
Específicamente, en una definición de una función **f** por recursión primitiva, **la recursión sólo se autoriza en la rama que corresponde a la definición de f en el caso S x**, y **las llamadas recursivas solo pueden usar el argumento x** (es decir, son siempre de la forma **f x**, llamando a la función en el número anterior del que se está considerando).

¿Es esta restricción muy severa? A su debido momento veremos que necesitaremos relajarla. Por ahora, sin embargo, nos dedicaremos a investigar el poder de la recursión primitiva **como método de programación**, tratando de aplicarla en la solución de varios problemas.

La gran ventaja de la recursión primitiva es que:

- Sabemos, sin necesidad de volver a demostrarlo cada vez, que **es una forma de recursión que funciona**, o sea, que efectivamente define funciones totales (que no se "cuelgan").

- El hecho de que una definición recursiva cumpla con la condición de estar hecha por recursión primitiva es **fácilmente verificable** por la mera observación del código correspondiente. Es decir, la verificación de esta condición es totalmente **mecánica** y hasta podría ser hecha por el compilador del lenguaje de programación.

Por lo tanto, si nos pudiéramos restringir a esta única forma de recursión tendríamos un criterio fijo para usar recursión que sería mecánicamente verificable y todas las definiciones serán bien fundadas, o sea, no existirán programas que no terminan.

Pero desafortunadamente es imposible dar una regla formal para formular definiciones recursivas que admita exactamente las que sean bien fundadas.
En otras palabras, cualquier modo de restringir la forma de las definiciones va a dejar afuera del lenguaje a una cierta clase que son bien fundadas.

Pero esto es tema de otro curso más avanzado... Lo que nos importa por ahora es decir que más adelante necesitaremos otras formas de recursión más complejas y, como consecuencia, la buena fundación de las definiciones dejará de ser automática y habrá que demostrarla caso a caso, considerando la particular forma de recursión que se haya empleado.

Por un tiempo viviremos en el mundo de la recursión primitiva. Y ésta tiene un poder considerable, como podrá verse a continuación.

En el siguiente ejemplo definiremos la función **doble :: N → N**, que calcula el doble de un número natural sin utilizar ninguna función auxiliar.
Otra vez planteemos el esquema de recursión primitiva para definirla:

doble = λn→ case n of { 0 → ?1 ; S x → ?2 }

Donde:
- ?1 debe ser el resultado de **doble 0**, que es **0.**
- ?2 debe ser el resultado de **doble (S x)**, asumiendo que sabemos calcular **doble x.**

Claramente, para pasar del doble de x al doble de S x, lo que debemos hacer es “sumar dos”, y eso no es otra cosa que agregar dos constructores S al doble de x.
Por lo tanto tenemos que **?2 = S(S(doble x)).**

La función doble, queda entonces definida como:

doble = λn→ case n of { 0 → 0 ; S x → S(S(doble x)) }

Computemos, por ejemplo el valor de doble (S(S 0)):

doble (S(S 0)) $\rightarrow_{\text{def. doble, β, case}}$ S(S(doble(S 0)))
$\rightarrow_{\text{def. doble, β, case}}$ S(S(S(S(doble 0))))
$\rightarrow_{\text{def. doble, β, case}}$ S(S(S(S 0))).

<u>Ejercicio</u>: definir la función **triple:: N → N**, que calcula el triple de un número natural sin utilizar ninguna función auxiliar.

Ahora que ya conocemos el tipo N de los Naturales y también los Booleanos, podemos programar funciones que hacen uso de ambos tipos para retornar resultados más útiles. El primer ejemplo de este tipo de funciones es la función **existe** que determina si existe un natural entre 0 y un número dado que cumpla con cierta condición.

El tipo de existe es el siguiente: **existe :: N → (N → Bool) → Bool**, y existe n p será True si existe algún natural entre 0 y n para el cual p es True.

Prestemos especial atención al **segundo** argumento: **(N → Bool)** es el tipo de cualquier función que evalúa una condición sobre un número natural y devuelve un Booleano (es decir, un **predicado**). Ejemplos de este tipo de funciones pueden ser la par o positivo definidas anteriormente.

La definición más simple de **existe** es la siguiente:

```
existe = λn p→ case n of {   0 → p 0  ;
                             S x → case p (S x) of { False → existe x p;
                                                     True → True }  }
```

Notar que, tal y como indica su tipo, la función recibe **dos parámetros, n** de tipo natural y, **p**, una función de **N → Bool** que usamos para calcular el resultado de **existe**.

En su caso base, cuando **n** es **0** la función devuelve **True** si el predicado **p** es **True** en **0**, y **False** si no (o sea, el resultado de evaluar el predicado **p** en **0**). En el caso recursivo en cambio, nos fijamos cuánto vale el predicado **p** en **S x:** si p (S x) es False, utilizamos la llamada recursiva para evaluar si existe un número entre 0 y x para el cual p es True. Si p (S x) es True, entonces ya sabemos que existe un número que cumple el predicado que buscábamos y devolvemos **True**.

Si queremos utilizar la función **existe** para saber si hay algún número natural par menor o igual que 3 podemos hacerlo de la siguiente manera: **existe (S (S (S 0))) par.**

Ejercicios:

1. Redefinir la función **existe** utilizando alguno de los conectivos booleanos vistos en clase para el caso S x.
2. Definir la función **todos :: N → (N → Bool) → Bool,** tal que **todos n p** devuelve True si todos los naturales entre 0 y n cumplen el predicado **p.**
3. Definir la función **contar :: N → (N → Bool) → N,** tal que **contar n p** devuelve la cantidad de números entre 0 y n que cumplen **p.**

De la misma forma que con los Booleanos, podemos crear funciones más interesantes al agregar más argumentos a nuestras funciones. Si pensamos en los números naturales y las operaciones que podemos hacer con ellos inmediatamente se nos ocurren las operaciones binarias **suma (+)** y el **producto (*).**

Veamos ahora cómo programarlos en Haskell.
Comenzaremos con la suma, que recibe dos números naturales y hace casos en uno de ellos, por ejemplo el primero:

```
(+) :: N → N → N
(+) = λm n→ case m of { 0 → ?1  ; S x → ?2 }
```

Observar que al hacer casos en **m,** estamos definiendo la función "**sumar m**" para un **n fijo**, que queda como un parámetro de la función suma.

Luego:

- ?1 debe ser el resultado de **sumar 0** a **n**, o sea: **n.**
- ?2 debe ser el resultado de **sumar(S x)** a **n.**

Esto no es evidente, pero **recordemos a la llamada recursiva** que nos permite suponer que **ya sabemos sumar x** a **n** (o sea, **sabemos calcular x + n**).
Ahora sí es fácil, ya que el resultado de (S x) + n no es otro que el siguiente de x + n, o sea **?2 = S(x + n).**

La suma queda definida entonces como:

```
(+) = λm n→ case n of { 0 → n  ; S x → S(x+n) }
```

Seguimos ahora con la multiplicación, recordemos que multiplicar, por ejemplo, 3*6 es simplemente sumar el número 6 una cantidad de veces: 3 en este caso.
Podemos entonces escribir 3 * 6 como: 6 + 6 + 6, que en realidad no es otra cosa que 3 * 6 = 6 + 2 * 6 = 6 + (6 + 1 * 6) = 6 + (6 + (6 + 0 * 6)) = 6 + (6 + (6 + 0)) que nos dará finalmente 18.

Con esto en mente, definimos el producto cómo:

```
(*) :: N → N → N
(*) = λm n→ case m of { 0 → 0  ; S x →  n + (x * n) }
```

Dejamos la **exponenciación** como **ejercicio** para el lector: la función **(^)** devuelve el resultado de multiplicar un número (base) por sí mismo un cantidad dada de veces.
Ej. **x ^ n = x * x * x * x … n veces**.

Otra funciones interesante que podemos definir con los números naturales son aquellas que operan dentro de un rango de números. Nos referimos aquí a las **sumatorias** y **productorias**. Nos vamos a centrar en aquellas que comienzan en cero y suman (o multiplican) todos los números desde cero hasta un límite dado.

El primer ejemplo, y muchas veces usado como ejemplo básico en la matemática discreta es la **sumatoria de todos los números naturales menores o iguales a n,** o sea, queremos en este caso programar lo siguiente:

$$\sum_{i=0}^{n} = 0 + 1 + 2 + 3 + \ldots + n$$

Llamamos a esta función **sumi** y su tipo es claramente **N → N.**

Otra vez, utilizando el esquema de recursión primitiva podemos expresar esto fácilmente en Haskell de la siguiente manera:

```
sumi :: N → N
sumi = λn→ case n of {0 → ?1; S x → ?2 }
```

Luego:

- **?1** es el resultado de $\sum_{i=0}^{0} i$**,** o sea **0**

-?2 es el resultado de $\sum_{i=0}^{Sx} i$**,** que debemos calcular **sabiendo calcular** $\sum_{i=0}^{x} i$**,** o sea

?2 = sumi x + S x.

¿Qué pasa ahora si queremos programar la siguiente expresión como una función en Haskell?

$$\sum_{i=0}^{n} 2 * i$$

En este caso queremos sumar el doble de cada número, podemos para ello hacer uso de la función doble definida anteriormente:

```
sumdobles :: N → N
sumdobles = λn→ case n of {0 → 0; S x → sumdobles x + doble (S x) }
```

En general, vamos a querer poder escribir funciones que representen la siguientes expresiones matemáticas:

$\sum_{i=0}^{n} f(i)$ donde f es una función aplicada a cada número (como por ejemplo doble)

Llamaremos a esta función: **sumfi.**

Es claro, por la forma de la expresión, que **f** es una función que **recibe un número natural y devuelve otro** (para poder sumarlo dentro de la sumatoria).

Entonces podemos definir **sumfi** de la siguiente manera:

sumfi :: N → (N → N) → N
sumfi = λn f. case n of {0 → f 0; S x → sumfi x f + f (S x) }

<u>Ejercicios</u>:

1. Haciendo uso de las funciones definidas anteriormente, defina una función que devuelva el resultado de sumar el triple de cada número entre 0 y un n dado.

2. Haciendo uso de las funciones definidas anteriormente, defina una función que se compute la siguiente expresión matemática: $\sum_{i=0}^{n} (3i + 1)$.
3. Programar la función **factorial :: N → N.**

Finalmente, cómo podemos programar en Haskell una función que sume todos los números **pares** entre 0 y n? Matemáticamente podemos escribir esto de la siguiente manera:

$\sum_{i=0}^{n} i \quad tal\ que\ i\ es\ par$

En Haskell podemos hacer esto de forma bastante sencilla, simplemente necesitamos conocer si cada número es par o no para sumarlo, y para ello vamos a usar la función **par** definida anteriormente.

sumpares :: N → N
sumpares = λn→ case n of { 0 → ?1; S x → ?2 }

- El caso base es trivial, 0 es un número par, por lo que lo agregamos a la suma: **?1 = 0**.

- Para el caso recursivo en cambio, necesitamos saber si **S x** es par o no, para ello podemos hacer uso de la función **par** definida arriba y una expresión **case**:

?2 = case par (S x) of {False → sumpares x ; True → sumpares x + S x}

Entonces:

sumpares = λn→ case n of { 0 → 0 ;
S x → case par (S x) of { False → sumpares x ;
True → S x + sumpares x }}

Con esta definición en mente, cómo podemos sumar todos los naturales que cumplan una condición genérica?

Ejercicio: programar la función **sumpi** que calcule:

$$\sum_{i=0}^{n} i \quad cuando\ se\ cumpla\ p(i)$$

para un predicado p. ¿Cuál es su tipo?