FUNDAMENTOS DE COMPUTACIÓN
Repartido: Bool - Parte 1

1. Tipos, expresiones y valores

Hasta el momento hemos dado reglas para construir expresiones con tipos, pero por ahora no hemos definido ningún tipo con datos efectivos. O sea, podemos construir sólo tipos funcionales, de la forma s → t, pero no tenemos ningún s ni t concretos...
A partir de ahora vamos a introducir tipos nuevos, pero antes de hacerlo, necesitamos dejar en claro qué significa ser un tipo, y qué conceptos involucra la definición de un tipo nuevo.

1) En primer lugar, conocer o **introducir un tipo** equivale a definir *qué cosa es un objeto de este tipo, y que significa que dos de esos objetos sean iguales por definición*.

2) En general, para cada tipo, introduciremos **expresiones** que nos permitan denotar a los objetos de ese tipo.

Es muy importante distinguir entre expresión y objeto de los diversos tipos:
Objetos son las entidades conceptuales (por ejemplos números, valores de verdad, funciones), mientras que las **expresiones** son entidades sintácticas (secuencias de caracteres).

La relación entre objetos y expresiones es la siguiente:
- Un objeto siempre tiene que ser denotado (representado) por alguna expresión .
- Para conocer cuál es el objeto representado por una expresión , se deben usar definiciones.

Por ejemplo, la expresión (6 x 12) + (9 x 8) denota un número. Pero para conocer qué número es este, se deben aplicar las definiciones de las operaciones de suma (+) y producto (x), a fin de **evaluar** la expresión . Por medio de este procedimiento se llegara a un valor (en este caso, 144).
Un **valor** es una expresión que no requiere ni puede ser evaluada, es decir, *debe ser entendida tal como se presenta* y **denota directamente** a un objeto de cierto tipo.

Resumiendo, todo objeto es denotado por una o más expresiones, y en general para llegar de una expresión al objeto representado, la expresión necesitará ser evaluada (computada) aplicando una serie de definiciones dadas explícitamente en el lenguaje.
Cuando se obtiene el valor de la expresión se llega a conocer el objeto denotado por la expresión.

2. Tipos con datos efectivos

Necesitamos tipos con datos efectivos. La idea más simple que se nos aparece para satisfacer esa necesidad es la de aquellos tipos cuyos objetos son dados explícitamente, por enumeración.
Necesariamente, la cantidad de esos objetos deberá ser finita, y por tanto llamaremos a los tipos en cuestión **tipos finitos**.

3. Los Booleanos

a) Definición del tipo y sus valores

Vamos a empezar con un tipo ya conocido por todos los que han utilizado algún lenguaje de programación: el tipo **Bool**, cuyos objetos son los valores de verdad ya conocidos (Falso y Verdadero).
Introducimos este tipo mediante las siguientes reglas (axiomas):

$$\frac{}{\textbf{Bool tipo}}$$

y sus valores False y True, que representan los valores de verdad anteriormente mencionados:

$$\frac{}{\textbf{False :: Bool}} \qquad \frac{}{\textbf{True :: Bool}}$$

Con estas reglas estamos diciendo que:
- Bool es un (nuevo) tipo
- Que tiene dos posibles valores: False y True.

*{en programación los valores de un tipo se llaman **constructores** y en Haskell, por convención, tanto los nombres de los tipos como los de los constructores deben comenzar con mayúscula}*

Hay numerosas funciones interesantes que operan sobre los booleanos, como por ejemplo las operaciones de la lógica clásica.
Antes de introducirlas, observemos primero que tenemos dos métodos para definir funciones sobre Bool:

Supongamos que deseamos definir una función **f :: Bool → t** para algún **tipo t cualquiera**. Esto puede hacerse de dos maneras:

La primera forma es por medio de una definición uniforme, que no distingue casos.
Un ejemplo de una definición de este tipo sería la función **identidad (idb)**, que devuelve el mismo valor que recibe, definida como sigue:

idb :: Bool → Bool
idb = λb→ b

Pero este mecanismo no nos permite definir funciones demasiado interesantes.

b) La construcción case

Lo que se requiere, es poder definir el valor de **f** en cada uno de los dos valores booleanos, es decir, cuánto valen **f** aplicada a **False** y **f** aplicada a **True**.

Por ello, introducimos el segundo mecanismo para definir funciones, que justamente nos permite **distinguir los dos casos** (False y True), dando explícitamente cada uno de los dos resultados requeridos. Este método se denomina en general, **análisis de casos**.
Para efectuar una definición por casos se necesita un nuevo dispositivo, que es la construcción **case**. En general, se escribirá:

f :: Bool → t
f = λb→ case b of { False → e1 ; True → e2 }

Donde **e1** y **e2** son los resultados deseados para la función **f** en los valores False y True respectivamente. Naturalmente, debe ser entonces **e1 :: t** y **e2 :: t**.

En el caso general, la construccion case opera sobre una expresión, que llamaremos **discriminante** del case, y permite distinguir casos o **ramas** para cada uno de sus posibles valores.

En el caso de arriba, el discriminante es b y sus valores posibles son False y True. De este modo, la construcción case permite definir la función distinguiendo los dos casos posibles de su argumento.

Un primer ejemplo es la **negación lógica:**

not :: Bool → Bool
not = λb→ case b of {False → True ; True → False }

Observar que lo que estamos diciendo en esta definición es que la función not recibe un booleano b, y dependiendo del valor de b devolverá:
- True, si b es False, y
- False, si b es True.

Formalmente, case es una construcción que permite definir expresiones y por lo tanto debe ser introducida por una regla que permita darle tipo a esas expresiones. La regla para construir expresiones con case es la siguiente

$$\frac{\textbf{e :: Bool} \qquad \textbf{e1 :: t} \qquad \textbf{e2 :: t}}{\textbf{case e of \{ False → e1 ; True → e2\} :: t}}$$

Las expresiones case son expresiones reducibles de nuestro lenguaje (redexes).
Para reducir una expresión de la forma case e of { False → e1 ; True → e2}, se debe reducir primero al argumento (e). Como e es de tipo Bool, su resultado será o bien False o bien True. Entonces, dependiendo del valor de e, el valor de la expresión será e1 o e2 respectivamente.

Esto justifica las siguientes igualdades:

case False of { False → e1 ; True → e2} = e1
case True of { False → e1 ; True → e2} = e2

Estas igualdades pueden leerse como reglas de cómputo, de izquierda a derecha.

c) Funciones Booleanas
Vamos ahora a programar algunas funciones, que son los conectivos de la lógica booleana. Empezamos con la conjunción (el "y", o "and"), que escribimos &&:

```
(&&) :: Bool → Bool → Bool
(&&) = λb1→ λb2→ case b1 of { False → False  ;  True → b2 }
```

En este caso estamos programando una función de dos argumentos. El método utilizado se va a repetir frecuentemente en lo sucesivo y consiste en elegir uno de ellos para realizar un análisis de casos, es decir considerar sus posibles valores en una expresión case.

Probablemente al intentar programar por primera vez la conjunción muchos escribirán la siguiente definición:

```
(&&) :: Bool → Bool → Bool
(&&) = λb1→ λb2→ case b1 of { False → False  ;
                              True → case b2 of { False → False  ;
                                                  True → True } }
```

Esta definición no es incorrecta, pero para el caso en que b1 es True, hace casos en b2... ¡para devolver luego exactamente el mismo valor que tiene b2!
Por eso es que preferimos la primera definición, que devuelve directamente el valor de b2 en el caso en que b1 tiene valor True.

Observar además, que hemos puesto el nombre de la función entre paréntesis: (&&). Esto se hace para poder usarla como un **operador infijo**, o sea, para aplicar la función (&&) a True y a False, podemos escribir True && False (con el nombre de la función en el medio de los operadores) en vez de (&&) True False, que sería la forma convencional de aplicar la función (&&) a esos dos valores.

Ahora vamos a programar algunos conectivos booleanos más:

- La disyunción lógica, o sea el o inclusivo, que escribimos ||:

(||) :: Bool → Bool → Bool
(||) = λb1→ λb2→ case b1 of { False → b2 ; True → True }

Observar que en el caso que los dos argumentos son True, el resultado de True || True es True (por eso se llama inclusivo).

- La implicación lógica (que usualmente escribimos ⇒), y es notada >> en Haskell.
Este conectivo es tal que b1 >> b2 es True siempre que: si b1 es True, entonces b2 también es True.
Otra forma de explicar esto es viendo que b1 >> b2 será False sólo en el caso en que b1 sea True y b2 False.
Esto lo podemos especificar haciendo una tabla de verdad, donde explicitamos todos los valores posibles de b1 y b2 y el resultado de b1 >> b2:

Tabla de Verdad del conectivo >>

b1	b2	b1 >> b2
False	False	True
False	True	True
True	False	False
True	True	True

La definición del conectivo >> será entonces:

(>>) :: Bool → Bool → Bool
(>>) = λb1→ λb2→ case b1 of { False → True ; True → b2}

- Finalmente, podemos definir la equivalencia lógica (notada ⇔), que se corresponde con la igualdad Booleana, y se define como el operador ==:

Tabla de Verdad del ==

b1	b2	b1 == b2
False	False	True
False	True	False
True	False	False
True	True	True

(==) :: Bool → Bool → Bool
(==) = λb1→ λb2→ case b1 of { False → not b2 ; True → b2}

Acá otra vez, algunos habrán programado esta función haciendo casos en b2 también:

(==) :: Bool → Bool → Bool
(==) = λb1→ λb2→ case b1 of { False → case b2 of {False → True ;
True → False};
True → case b2 of {False → False;
True → True}}

Observar que ambas definiciones son equivalentes.

4. Nuevas Igualdades y Redexes

Cada una de las funciones que programamos fue dada mediante una definición.
Todas las definiciones que hagamos son igualdades que iremos agregando a nuestro lenguaje, y serán consideradas redexes (leídas de izquierda a derecha).

Así, tendremos reducciones del estilo:

not False $\rightarrow_{\text{def. not}}$ (λb → case b of {False → True ; True → False }) False
$\rightarrow_{\beta}$ case False of {False → True ; True → False }
$\rightarrow_{\text{case}}$ True.

True == (not True)
= (notación infija)
(==) True (not True)
$\rightarrow_{\text{def. ==}}$ (λb1→ λb2→ case b1 of { False → not b2 ; True → b2}) True (not True)
$\rightarrow_{\beta} \rightarrow_{\beta}$ case True of { False → not (not True) ; True → not True}
$\rightarrow_{\text{case}}$ not True
$\rightarrow_{\text{def. not}}$ (λb→ case b of {False→ True ; True → False }) True
$\rightarrow_{\beta}$ case True of {False → True ; True → False }
$\rightarrow_{\text{case}}$ False.

Estas son ejecuciones a mano de nuestros programas. Partimos de una expresión compleja (como True == not True) y llegamos a su **valor** reduciendo los redexes que aparecen en ella.
Todas estas reducciones las va a hacer el intérprete de nuestro lenguaje de programación, estos ejemplos son para entender cómo funcionan las reducciones.

5. Resumiendo: Hasta ahora tenemos las siguientes reglas en nuestro lenguaje de programación:

a) Para formar tipos:

$$\frac{s \text{ tipo} \quad t \text{ tipo}}{s \to t \text{ tipo}}$$

$$\frac{}{\text{Bool tipo}}$$

b) Para formar expresiones

$$\frac{}{x :: t \; [x :: t]}$$

$$\frac{f :: s \to t \quad e :: s}{f\ e :: t}$$

$$\frac{e :: t \; [\, x :: s \,]}{\lambda x \to e :: s \to t}$$

$$\frac{}{\text{False} :: \text{Bool}}$$

$$\frac{}{\text{True} :: \text{Bool}}$$

$$\frac{e :: \text{Bool} \qquad e1 :: t \qquad e2 :: t}{\text{case } e \text{ of } \{ \text{False} \to e1 \; ; \; \text{True} \to e2 \} :: t}$$

c) Igualdades y Redexes

Igualdad Alfa: $\lambda x \to e =_{\alpha} \lambda y \to (e[x := y])$, con y no libre en e.

Igualdad Beta: $(\lambda x \to e)\ a =_{\beta} e[x := a]$

Reducción Beta (Redex): $(\lambda x \to e)\ a \to_{\beta} e[x := a]$

Igualdades Case: $\text{case False of } \{ \text{False} \to e1 \; ; \; \text{True} \to e2 \} =_{case} e1$

$\text{case True of } \{ \text{False} \to e1 \; ; \; \text{True} \to e2 \} =_{case} e2$

Reducciones Case (Redexes): $\text{case False of } \{ \text{False} \to e1 \; ; \; \text{True} \to e2 \} \to_{case} e1$

$\text{case True of } \{ \text{False} \to e1 \; ; \; \text{True} \to e2 \} \to_{case} e2$

Además, todas las definiciones que hagamos son igualdades que iremos agregando a nuestro lenguaje, y serán consideradas redexes en las computaciones.