Programación con tipos dependientes en Idris 2

ADTs I: Bool

Axel Suárez Po

April 23, 2021

BUAP

Contenidos

Introducción

Una pregunta interesante

¿Cuántas funciones *distintas* hay de

`Bool -> Bool`?

```
f : Bool -> Bool
f b = ?r
```

```
boolean f(boolean b) {
  return ?;
}
```

Listing 1: Bool a Bool en Idris y Java

[1] Trate de contestar en menos de 10 segundos

Las funciones obvias

```haskell
-- La función not
notb : Bool -> Bool
notb True  = False
notb False = True


-- La función identidad
idb : Bool -> Bool
idb True  = True
idb False = False
```

```java
boolean notb(boolean b) {
  if (b == true)
    return false;
  else
    return true;
}

boolean idb(boolean b) {
  if (b == true)
    return true;
  else
    return false;
}
```

Listing 2: Funciones Bool a Bool obvias.

[1] Claramente se pueden escribir de forma más concisa.

Las funciones no tan obvias

```haskell
-- La función constante True
constTrue : Bool -> Bool
constTrue True  = True
constTrue False = True


-- La función constante False
constFalse : Bool -> Bool
constFalse True  = False
constFalse False = False
```

```java
boolean constTrue(boolean b) {
  if (b == true)
    return true;
  else
    return true;
}

boolean constFalse(boolean b) {
  if (b == true)
    return false;
  else
    return false;
}
```

Listing 3: Funciones Bool a Bool no tan obvias.

[1] ¿Serán todas? ¿Cómo estar seguro?

Una pregunta más interesante

¿Cuántas funciones hay de **`Bool -> Bool -> Bool`**?

```
f : Bool -> Bool -> Bool
f a b = ?r
```

```
boolean f(boolean a, boolean b) {
  return ?;
}
```

Listing 4: Dos Bool's a Bool en Idris y Java

Tipos de datos algebraicos

¿Qué es un tipo de datos algebraico?

ADT

Un tipo de datos algebraico (ADT) es un tipo de datos **compuesto**, esto quiere decir que se forma combinando otros tipos de datos.

Las dos clases más comunes de tipos de datos compuestos utilizados son:

- **Productos:** Los valores de este tipo son combinaciones de valores de los tipos que lo componen. (**Tuplas**)
- **Sumas:** Los valores de este tipo alternan entre tipos distintos de forma excluyente. (**Either**)

Sumas

Los **tipos suma** (*sum types*) también suelen nombrarse como **coproductos, uniones disjuntas (etiquetadas), variantes o alternativas.**

La sintaxis para definir un tipo suma, es separando sus **constructores** con barras verticales (|).

Un ejemplo de estos, es un tipo bastante conocido: **`Bool`**.

```
data Bool = False | True
```

Listing 5: Definición de Bool de la librería estándar.

[1] Más adelante se definirá con precisión a que se refiere la palabra *constructores*.

Sumas: Construcción

Para *construir* un valor que pertenezca a este tipo se hace uso de sus constructores:

```
data Bool = False | True

x : Bool
x = True

x' : Bool
x' = False
```

Listing 6: Valores del tipo Bool.

[1] Nótese otra vez que se está siendo *vago* con la definición de qué quiere decir construir.

Sumas: Utilización

Cuando se tiene una expresión de un tipo suma, hay una cantidad de cosas limitadas que se pueden hacer:

- Ignorar el valor.
- Colocarlo en **uno o más lugares** donde se espera una expresión de ese tipo.
  - Pasarlo como parámetro
  - Referenciarlo en expresiones con el mismo tipo
- *Deconstruirlo* con **múltiples** *patrones de constructor*, utilizando el **pattern matching**.

Las dos primeras opciones son comunes a todas las expresiones, pero la última es única de los tipos suma.

Pattern matching

Cuando se utiliza **pattern matching**, se busca “encajar” (*match*) un valor con ciertos patrones, pudiendo asignar variables cuando este *match* es exitoso.

Este pattern matching puede ocurrir en diferentes posiciones, pero nos enfocaremos por el momento en la **definición de funciones**.

El pattern matching puede tomar varias formas:

- Patrón de literales
- **Patrón de constructor**
- Patrón de variable y patrón de descartamiento (*catch all*)
- Patrones especializados: **as-patterns, dot-patterns, view-patterns, with-patterns, etc**.

Pattern matching: Patrón de constructor

El pattern matching en el que centraremos nuestra atención es el **patrón de constructor.**

Tal como su nombre lo indica, consiste en enumerar los constructores del tipo que se está operando:

```haskell
data Bool = False | True

-- La función not
notb : Bool -> Bool
notb True  = False
notb False = True

-- La función identidad
idb : Bool -> Bool
idb True  = True
idb False = False
```

Listing 7: Patrón de constructor

Pattern matching: Patrón de variable

Sin embargo, la función identidad se puede definir de una manera más sencilla.

Para simplemente capturar el valor se puede utilizar un identificador (que no se encuentre en otro patrón):

```haskell
data Bool = False | True

idb : Bool -> Bool
idb True  = True
idb False = False

idb' : Bool -> Bool
idb' b = b

-- Cualquier identificador válido
idb'' : Bool -> Bool
idb'' nombreLargo = nombreLargo
```

Listing 8: Patrón de variable

Pattern matching: Patrón de descartamiento

Existen ocasiones en las que no es necesario utilizar un parámetro, como en las funciones constantes que se encontraron con anterioridad:

```haskell
data Bool = False | True

constTrue : Bool -> Bool
constTrue True  = True
constTrue False = True

-- Otra alternativa
constTrue'' : Bool -> Bool
constTrue'' b = True

-- Una forma más concisa
constTrue' : Bool -> Bool
constTrue' _ = True
```

Listing 9: Patrón de descartamiento

Pattern matching: Catch-all

Los dos tipos anteriores de pattern matching (variable y descartamiento) tienen un comportamiento que se conoce como **catch-all**.

Esto quiere decir que cuando se encuentra uno de estos, los siguientes patrones sobre este mismo parámetro se vuelven inalcanzables.

```haskell
-- Uso incorrecto de un patrón catch-all
-- compila con advertencias
constTrue'alt : Bool -> Bool
constTrue'alt _     = True
constTrue'alt False = False
```

Listing 10: Patrón no alcanzable

Totalidad

Los patrones catch-all son particularmente útiles para asegurar la **totalidad** de las funciones.

Por ejemplo, el tipo de datos **`Char`** lo podemos conceptualizar como un tipo suma de todos los carácteres posibles, por lo que una función **`Char -> String`** requeriría bastantes patrones.

```haskell
-- Esta definición solo es conceptual
data Char = 'a' | 'b' | ...

inicialANombre : Char -> String
inicialANombre 'a' = "Axel"
inicialANombre 'i' = "Iván"
inicialANombre 'j' = "Jesús"
inicialANombre 's' = "Samuel"
inicialANombre  _  = "Otro"
```

Listing 11: Catch-all para función total

Parcialidad

Si no se han cubierto todas las alternativas y además no se incluye un patrón catch all, entonces se dice que la función es **parcial**.

Existen dos casos en los que se tienen funciones parciales:

- **Error del programador:** Al programador le faltó considerar alternativas para la función, por lo que deberá agregar un patrón catch-all o manejar los demás casos.
- **Se requiere una función parcial:** En el casos de que realmente se requiera una función parcial, Idris requiere que se anote explícitamente como tal con la palabra clave `partial`.

GADT's

La notación de GADT's es una alternativa para definir tipos de datos de una forma más explícita.

Además, permite definir tipos de datos **indexados** y **parametrizados**.

El tipo de datos **`Bool`** también se puede definir como sigue:

```
data Bool : Type where
  False : Bool
  True  : Bool

-- La definición anterior
data Bool = False | True
```

Listing 12: GADT para Bool

GADT's

Aparte de ser más explícita y tener mayores capacidades, también permite ver claramente que es lo que implica definir un tipo de datos:

- **`Bool`** es un tipo
- **`False`** es del tipo **`Bool`**
- **`True`** es del tipo **`Bool`**
- No hay otra cosa que sea un **`Bool`**

Esta última regla es lo que garantiza que un pattern matching exhaustivo implique la totalidad de una función.

```
data Bool : Type where
  False : Bool
  True  : Bool
```

Listing 13: GADT para Bool

Tipos suma, otra vez.

Se ha mencionado que a estos se les conoce como tipos suma, pero no se ha mostrado un ejemplo de suma de tipos.

A continuación se muestra una:

```
data RGB = Red | Green | Blue
data CMY = Cyan | Magenta | Yellow

data Color = R RGB | C CMY
```

Listing 14: ADT para Colores

Como se puede observar, se han definido dos tipos de datos y se ha creado una suma de ellos.

Tipo suma: Colores

Si se quisiera definir una función sobre este tipo de datos, se puede realizar pattern matching recursivo sobre sus constructores:

```
complemento : Color -> Color
complemento (R Red) = C Cyan
complemento (R Green) = C Magenta
complemento (R Blue) = C Yellow
complemento (C Cyan) = R Red
complemento (C Magenta) = R Green
complemento (C Yellow) = R Blue
```

Listing 15: Función sobre tipo de datos más complejo

Tipo suma: Bool

Bool no parece ser un tipo suma por la forma en que está conformado, sin embargo, utilizando el tipo **Unit**, se puede ver que es la unión etiquetada de **Unit** consigo mismo.

```haskell
data Bool' = T () | F ()

-- Isomorfismo Bool y Bool'
from : Bool -> Bool'
from True  = T ()
from False = F ()

to : Bool' -> Bool
to (T _) = True
to (F _) = False
```

Listing 16: Isomorfismo Bool

Unit

`Unit` es un tipo de datos primitivo que sólo tiene un constructor que no puede contener información.

El tipo de datos **`Unit`** se define como sigue:

```
data () = ()
-- Alternativamente
data Unit = Unit
```

Listing 17: Tipo de datos Unit

Estas definiciones son especiales en Idris y no se pueden replicar desde el código del usuario.

Unit

Finalmente, es importante resaltar que las uniones etiquetadas de tipos se pueden realizar entre cualesquiera tipos.

```
data Numero = NS String | NI Int

data App = Pure Int | Fn (Int -> Int)
```

Listing 18: Unión de tipos