

Buenos tipos para el desarrollo web

Alejandro Serrano

29 de febrero de 2016

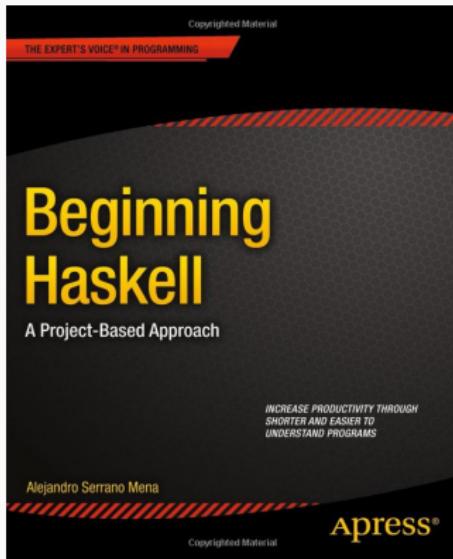
Haskell-MAD

¿De qué va esta charla?

Una pausa para la publicidad

Hola, me llamo Alejandro

- Nací y he crecido en Madrid
- Comencé a interesarme por Haskell allá por el 2009
- He participado dos veces en el Google Summer of Code mejorando el soporte de Haskell para Eclipse y Emacs
- El año pasado publiqué un libro, *Beginning Haskell*



Vivo en Utrecht (Países Bajos)

- En la Universidad tienen un grupo especializado en prog. funcional
- Me fui de Erasmus en 2010/11 para aprender
- Y ahora vivo allí y me estoy doctorando :)



Todos los veranos organizamos un curso de 2 semanas:
Applied Functional Programming
(este año es del 4 al 15 de julio)

Jugando con tipos (sin quemarnos)

Tipos de datos algebraicos

```
data Numero = Cero  
          | Suc Numero
```

```
data Arbol a = Hoja a  
          | Nodo (Arbol a) a (Arbol a)
```

```
*Charla> :t Suc (Suc Cero)
```

```
Suc (Suc Cero) :: Numero
```

```
*Charla> :t Nodo (Hoja 'a') 'b' (Hoja 'c')
```

```
Nodo (Hoja 'a') 'b' (Hoja 'c') :: Arbol Char
```

Declarando el tipo de los constructores

```
data Numero = Cero | Suc Numero
```

```
data Arbol a = Hoja a | Nodo (Arbol a) a (Arbol a)
```

```
{-# LANGUAGE GADTs #-}
```

```
data Numero where
```

```
  Cero :: Numero
```

```
  Suc :: Numero → Numero
```

```
data Arbol a where
```

```
  Hoja :: a → Arbol a
```

```
  Nodo :: Arbol a → a → Arbol a → Arbol a
```

Describiendo campos de una tabla

```
data CampoUsuario where
    Nombre :: String          → CampoUsuario
    Apellidos :: String        → CampoUsuario
    Edad     :: Integer         → CampoUsuario
    Direccion :: String → String → CampoUsuario
```

Describiendo campos de una tabla

```
data CampoUsuario where
    Nombre :: String          → CampoUsuario
    Apellidos :: String        → CampoUsuario
    Edad     :: Integer         → CampoUsuario
    Direccion :: String → String → CampoUsuario
```

¿Qué tipo le damos a la siguiente función?

```
*Charla> getCampoUsuario (Nombre "Alejandro")
"Alejandro" :: String
*Charla> getCampoUsuario (Edad 27)
27           :: Integer
```

ADTs generalizados (GADTs)

Los constructores pueden *especializar* el tipo que construyen

```
data CampoUsuario a where
  Nombre :: String          → CampoUsuario String
  Apellidos :: String        → CampoUsuario String
  Edad    :: Integer          → CampoUsuario Integer
  Direccion :: String → String → CampoUsuario (String, String)
```

ADTs generalizados (GADTs)

Los constructores pueden *especializar* el tipo que construyen

```
data CampoUsuario a where
  Nombre :: String          → CampoUsuario String
  Apellidos :: String        → CampoUsuario String
  Edad    :: Integer         → CampoUsuario Integer
  Direccion :: String → String → CampoUsuario (String, String)
```

Ahora ya podemos tipar la función correctamente :)

```
getCampoUsuario :: CampoUsuario t → t
```

Asegurando estáticamente la validación

Con el ánimo alto, decidimos usar esta técnica para diferenciar cadenas HTML validadas y no validadas contra inyección de código

- Esta técnica se denomina *tipos fantasma* (*phantom types*)

data CadenaHTML v where

NoV :: String → CadenaHTML NoValidada

V :: String → CadenaHTML Validada

-- Con esto eliminamos los elementos dudosos

escapar :: CadenaHTML NoValidada → CadenaHTML Validada

-- Sólo podemos mostrar cadenas escapadas

mostrar :: CadenaHTML Validada → HTML

¿De dónde sacamos los índices?

Una opción es definir tipos de datos vacíos

```
{-# LANGUAGE EmptyDataDecls #-}  
data NoValidada  
data Validada
```

El problema es que sólo *NoValidada* y *Validada* tienen sentido como índices de *CadenaHTML*, pero GHC no restringe ese hecho

```
*Charla> :k CadenaHTML Bool  
CadenaHTML Bool :: *
```

Por los cerros de ÚbeKa

Los *kinds* son los tipos de los tipos

- El kind * es el de los tipos de valores

```
*Charla> :k Integer
```

```
Integer :: *
```

```
*Charla> :k CadenaHTML Validada
```

```
CadenaHTML Validada :: *
```

- Los constructores de tipo tienen un kind con flechas

```
*Charla> :k Arbol
```

```
Arbol :: * -> *
```

```
*Charla> :k CadenaHTML
```

```
CadenaHTML :: * -> *
```

Promocionando tipos

En GHC podemos pedir que por cada tipo de datos se genere un nuevo kind con un habitante por constructor

```
{-# LANGUAGE DataKinds, KindSignatures #-}  
data EstadoValidacion = Validada | NoValidada  
data CadenaHTML (v :: EstadoValidacion) where ...
```

Es *como si* pudiésemos usar valores en los índices

Promocionando tipos

En GHC podemos pedir que por cada tipo de datos se genere un nuevo kind con un habitante por constructor

```
{-# LANGUAGE DataKinds, KindSignatures #-}  
data EstadoValidacion = Validada | NoValidada  
data CadenaHTML (v :: EstadoValidacion) where ...
```

Es *como si* pudiésemos usar valores en los índices

```
*Charla> :k CadenaHTML Bool  
<interactive>:1:12:  
    The first argument of 'CadenaHTML'  
        should have kind 'EstadoValidacion',  
        but 'Bool' has kind '*'
```

Listas indizadas por longitud

¿Cómo definimos este tipo de datos?

```
*Charla> :t 'a' ::/ 'b' ::/ 'c' ::/ Fin  
'a' ::/ 'b' ::/ 'c' ::/ Fin  
:: Lista ('Suc ('Suc ('Suc 'Cero))) Char
```

Listas indizadas por longitud

¿Cómo definimos este tipo de datos?

```
*Charla> :t 'a' :: 'b' :: 'c' :: Fin  
'a' :: 'b' :: 'c' :: Fin  
:: Lista ('Suc ('Suc ('Suc 'Cero))) Char
```

Usando GADTs con un tipo promocionado de índice

```
data Numero = Cero | Suc Numero  
infixr 8 :::  
data Lista (n :: Numero) e where  
  Fin ::                               Lista Cero   e  
  (::) :: e → Lista n e → Lista (Suc n) e
```

Tipando algunas funciones

primero :: Lista (Suc n) e → e

-- primero Fin = ??

primero (e :/: _) = e

Tipando algunas funciones

primero :: Lista (Suc n) e → e

-- primero Fin = ??

primero (e :/: _) = e

```
*Charla> head []
```

```
*** Exception: Prelude.head: empty list
```

```
*Charla> primero Fin
```

```
<interactive>:27:9:
```

```
Couldn't match type 'Cero with 'Suc n0
```

Tipando algunas funciones

primero :: Lista (Suc n) e → e

-- primero Fin = ??

primero (e :/: _) = e

```
*Charla> head []
```

```
*** Exception: Prelude.head: empty list
```

```
*Charla> primero Fin
```

```
<interactive>:27:9:
```

```
Couldn't match type 'Cero with 'Suc n0
```

unir :: Lista n e → Lista m e → Lista ?? e

unir Fin lst = lst

unir (e :/: r) lst = e :/: unir r lst

Los tipos también forman familias

Una *familia de tipos* es una función a nivel de tipos

type family $n :+ m$ **where**

Cero $:+ m = m$

Suc $n :+ m = Suc(n :+ m)$

Ahora sí que podemos darle un tipo a *unir*

unir :: *Lista n e* \rightarrow *Lista m e* \rightarrow *Lista (n :+ m) e*

Los tipos también forman familias

Una *familia de tipos* es una función a nivel de tipos

type family $n :+ m$ **where**

Cero $:+ m = m$

Suc $n :+ m = Suc(n :+ m)$

Ahora sí que podemos darle un tipo a *unir*

unir :: *Lista n e* \rightarrow *Lista m e* \rightarrow *Lista (n :+ m) e*

Nótese que GHC no conoce ninguna propiedad aritmética

unir :: *Lista n e* \rightarrow *Lista m e* \rightarrow *Lista (m :+ n) e*

Charla.hs:

```
Could not deduce (m ~ (m :+ 'Cero))
```

Charla.hs:

```
Could not deduce ((m :+ 'Suc n) ~ 'Suc (m :+ n))
```

Tipos asociados

Las familias de tipos también pueden aparecer *asociadas* a clases

- En ese caso forman una familia *abierta*

class Coleccion *c* **where**

type Elemento *c*

vacio :: *c*

(//) :: Elemento *c* → *c* → *c*

instance Coleccion [*e*] **where**

type Elemento [*e*] = *e*

vacio = []

(//) = (:)

instance Coleccion IntSet **where**

type Elemento IntSet = Int

...

En resumen

Haskell tiene un gran soporte para programación a nivel de tipo

- Los *kinds* son los tipos de los tipos
- Podemos crear nuevos kinds por *promoción*
- Y definir funciones sobre tipos usando *familias*
- Los *GADTs* nos permiten refinar el tipo de los constructores

Tipos más concretos = más errores capturados al compilar

¿Qué no puedo hacer en Haskell?

Los valores pasados a una función *no pueden influenciar* su tipo

- No puedo escribir una función que dado un número n y un valor v , devuelva una *Lista n e* repitiendo en valor v
- Existen dos mundos separados: valores y tipos/kinds/...
- De ahí que sea necesaria la promoción

Tipos dependientes

Los sistemas de *tipos dependientes* eliminan esa barrera

- Algunos lenguajes que los soportan son **Idris**, Agda y Coq
- Ojo, son lenguajes experimentales

Además, se pueden usar para *probar* propiedades de los programas

- Por ejemplo, *invertir (invertir lst) = lst*

repetir : (n : Numero) → e → Lista n e

repetir Cero x = Fin

repetir (Suc n) x = x ∷: repetir n x

Estoy en ascuas, quiero saber más

Vídeos (los podéis encontrar en YouTube)

- *Depending on Types* de Stephanie Weirich
 - Keynote de ICFP 2014
- *A practical intro to Haskell GADTs* de Richard Eisenberg
 - Parte de LambdaConf (este año en mayo)
- *Adventures with Types* de Simon Peyton Jones
 - Un conjunto de charlas sobre el sistema de tipos de Haskell

Libros (perfectos para leer en el tren)

- *Beginning Haskell*, por supuesto :P
- *Type-Driven Development with Idris* de Edwin Brady

¡Desarrollo web!

Un poco de relax con JSON

JSON está por todas partes

En estos tiempos modernos, no hay aplicación web que se precie que no consuma y genere JSON (aunque sea un poquito)

- JSON = Javascript Object Notation

```
{  
    "nombre"      : "Alejandro",  
    "apellidos"   : "Serrano Mena"  
    "edad"        : 27  
}
```

- Trabajar con JSON es tedioso y propenso a errores tontos

JSON está por todas partes

En estos tiempos modernos, no hay aplicación web que se precie que no consuma y genere JSON (aunque sea un poquito)

- JSON = Javascript Object Notation

```
{  
    "nombre"      : "Alejandro",  
    "apellidos"   : "Serrano Mena"  
    "edad"        : 27  
}
```

- Trabajar con JSON es tedioso y propenso a errores tontos

Esta parte va sobre cómo *no* escribir código que trate JSON

- Las mejores líneas de código son las que no escribes ;)

Esón, padre de Jasón

aeson es el estándar de facto para trabajar con JSON en Haskell

```
{  
    "nombre"      : "Alejandro",  
    "apellidos"   : "Serrano Mena"  
    "edad"        : 27  
}
```

```
{-# LANGUAGE OverloadedStrings #-}  
object ["nombre"     .= String "Alejandro"  
       , "apellidos"   .= String "Serrano Mena"  
       , "edad"        .= Number 27]
```

De tu tipo a JSON y viceversa

Es preferible, no obstante, trabajar con tus propios tipos de datos

- Si trabajamos con *Value* directamente, perdemos las ventajas de un tipado fuerte
- En la frontera de nuestra aplicación con el mundo exterior, no queda más remedio que convertir de y a JSON

class *ToJSON* where

toJSON :: *a* → *Value*

class *FromJSON* where

parseJSON :: *Value* → *Parser a*

-- Parser es como un Maybe más informativo

Tedio, aburrimiento, sopor

```
data Persona = Persona { nombre :: String
                        , apellidos :: String
                        , edad      :: Integer }
deriving (Eq, Show)

instance ToJSON Persona where
  toJSON (Persona n a e) = object ["nombre" .= n
                                    , "apellidos" .= a
                                    , "edad" .= e]

instance FromJSON Persona where
```

parseJSON v = ...

¡Y esto por cada tipo de datos en tu aplicación!

Derivación genérica al rescate

```
{-# LANGUAGE DeriveGeneric #-}  
import Data.Aeson  
import GHC.Generics  
data Persona = Persona { ... }  
    deriving (Eq, Show, Generic)  
instance ToJSON Persona  
instance FromJSON Persona
```

¡No es necesario escribir más código!

- Las instancias se derivan de la propia declaración de tipo
- Los nombre de campos se usan como claves

Hora de un merecido descanso

Sirviendo con Servant

La API de tu aplicación web es su tipo

La API de tu aplicación web es su tipo

- Especifica el contrato y los formatos aceptados por cada ruta
- Independientemente de la conversión y el transporte
- Una misma API da lugar a diferentes implementaciones
 - Servidor
 - Cliente
 - Documentación

Ejemplo: un servicio de contadores

- PUT /new: genera un nuevo contador
 - El cuerpo de la llamada indica el nombre del contador
 - Devuelve un identificador numérico
- POST /step/:id: aumenta en uno el contador
- GET /value/:id: obtiene el valor del contador
- GET /list: lista los contadores
 - Opcionalmente puede llevar un parámetro order que indica si ordenar por orden creciente o decreciente de valor

Nuestro tipo-API

```
type NewAPI = "new"    > ReqBody '[PlainText] Text
              > Put '[JSON] CounterId
type StepAPI = "step"  > Capture "id" CounterId
              > Post '[JSON] Integer
type ValueAPI = "value" > Capture "id" CounterId
              > Get '[JSON] Integer
type ListAPI = "list"   > QueryParam "order" Order
              > Get '[JSON] [Counter]
type AppAPI = NewAPI :||> StepAPI :||> ValueAPI :||> ListAPI
```

Locura en los tipos

Se usan muchísimos tipos promocionados

- Cadenas a nivel de tipo, cuyo kind es *Symbol*
- Listas a nivel de tipo, encabezadas por ‘
 - Por ejemplo, ‘[*Int*, *String*] :: ‘[*]
 - Nótese la diferencia entre [*String*] y ‘[*String*]

Construyendo rutas

- `:</>` combina rutas para construir una aplicación
- `>` combina segmentos para construir una ruta

Hay multitud de posibilidades para un segmento

- "cadena" indica un segmento que ha de aparecer literalmente
- Hay varias formas de capturar información
 - *Capture "campo" Tipo* indica que lo que aparezca en ese segmento estará disponible luego bajo el nombre *campo*
 - *ReqBody '[Formatos]' Tipo* indica que el cuerpo de la petición contendrá un valor serializado en uno de los formatos indicados
 - *QueryParam "campo" Tipo* se usa para ?campo=valor
- *Verbo '[Formatos]' Tipo* indica una acción HTTP
 - Por ejemplo *Get*, *Post*, *Put*, *Delete* o *Patch*
 - Tal acción tendrá como resultado un valor del tipo mencionado
 - Que puede ser servido en uno de los formatos indicados

Preparando el percal

```
type CounterId = Integer
data Counter = Counter { counterId :: CounterId
                        , counterName           :: Text
                        , counterValue :: Integer }
deriving (Eq, Show, Generic)
instance FromJSON Counter
instance ToJSON Counter
```

Cómo mantener el estado de la aplicación

Para guardar el estado de la aplicación vamos a usar STM

- STM = *Memoria Software Transaccional*
- *TVar* = variable con acceso arbitrado transaccionalmente
- Protección frente acceso concurrente

Su uso es **muy** sencillo

- Creamos *TVars* con *newTVar* o *newTVarIO*
- Escribimos *scripts* transaccionales dentro de la mónada *STM*
 - Usando *readTVar*, *writeTVar* o *modifyTVar*
- Ejecutamos un *script* usando *atomically*

Implementando un trocito de API

Para servir *API* tenemos que implementar un valor *Server API*

- *Server* es una familia de tipos que indica qué necesitamos definir para servir una ruta
- Todas las variables capturadas se convierten en argumentos
- Este servicio vive en el paquete `servant-server`

```
*Charla> :kind! Server NewAPI
Server NewAPI :: *
= Text -> EitherT ServantErr IO Integer
```

- En un servidor podemos ejecutar acciones *IO*
 - Tenemos que usar *liftIO* para ello

Haskell es el mejor lenguaje imperativo

```
type Estado = (TVar CounterId, TVar [Counter])  
servirNewAPI :: Estado → Server NewAPI  
servirNewAPI (ultimoid, lista) nombre = liftIO $  
    atomically $ do modifyTVar ultimoid (+1)  
        nuevold ← readTVar ultimoid  
        let c = Counter nuevold nombre 0  
        modifyTVar lista (c:)  
        return nuevold
```

Ejecutando el servidor

Usando `serve` convertimos un *Server* en una *Application* de WAI

- WAI = Web Application Interface
- Protocolo común a todas las aplicaciones web Haskell
 - Similar a servlets de Java o a WSGI para Python

Para correr una *Application* necesitamos un *runner*

- Warp es el más común y funciona muy bien

Ejecutando el servidor

Usando `serve` convertimos un *Server* en una *Application* de WAI

- WAI = Web Application Interface
- Protocolo común a todas las aplicaciones web Haskell
 - Similar a servlets de Java o a WSGI para Python

Para correr una *Application* necesitamos un *runner*

- Warp es el más común y funciona muy bien

servidor :: *Estado* → *Application*

servidor s = `serve (Proxy :: Proxy NewAPI) (servirNewAPI s)`

main :: *IO ()*

main = **do** *estado* ← (,) $\$$ *newTVarIO* 0 $\langle *\rangle$ *newTVarIO* []
run 8081 (*servidor estado*)

Implementando el listado de contadores

servirListAPI :: Estado → Server ListAPI

servirListAPI (_, lista) _ = liftIO \$ readTVarIO \$ lista

servidor :: Estado → Application

servidor s = serve (Proxy :: Proxy (NewAPI :<|> ListAPI))
(servirNewAPI s :<|> servirListAPI s)

Implementando el listado de contadores

servirListAPI :: Estado → Server ListAPI

servirListAPI (_, lista) _ = liftIO \$ readTVarIO \$ lista

servidor :: Estado → Application

servidor s = serve (Proxy :: Proxy (NewAPI :<|> ListAPI))
(servirNewAPI s :<|> servirListAPI s)

No sé a vosotros, a mí esto me parece *magia* 

Devolviendo HTML

Ahora queremos devolver HTML además de JSON

1. Añadir el nuevo formato a la lista de *Get*

```
type ListAPI = "list" > QueryParam "order" Order  
                      > Get '[JSON, HTML] [Counter]
```

2. Implementar la conversión a *HTML* mediante *ToHtml*

```
instance ToHtml [Counter] where  
    toHtml = ...
```

Generando HTML con Haskell

Hay 3 grandes posturas para generar HTML

- Un lenguaje de *templates* ajeno a Haskell
 - Por ejemplo, `mustache`
- Un lenguaje integrado en Haskell (un DSL)
 - Por ejemplo, `blaze-html` y `lucid`
 - Si preferimos usar Markdown, `markdown`
- Un término medio: un lenguaje ajeno a Haskell que en tiempo de *compilación* se convierte en llamadas a una librería
 - Usando *quasi-quoting* para parsear
 - `shakespeare` es el mayor exponente

Generando HTML con lucid

- Los elementos y atributos son funciones
 - El nombre es el mismo que en HTML más _
- El contenido de una etiqueta o el valor de un atributo se da como *argumento* de la función
- Los atributos de una etiqueta como una lista
- Si dentro de un elemento hay varios elementos, se puede usar
 - El combinador monoidal <>
 - Notación monádica con **do**

```
div_[class_ "info"] $ do      <div class="info">  
  p_(strong_ "Lugar")          <p><strong>Lugar</strong></p>  
  p_ "Haskell-MAD"           <p>Haskell-MAD</p>  
                                </div>
```

Ventajas de ser una mónada

```
{-# LANGUAGE FlexibleInstances #-}  
import Lucid  
instance ToHtml [Counter] where  
  toHtml cs = ul_ $ forM_ cs $ λ(Counter _ nm v) → do  
    li_ $ do strong_ (toHtml nm)  
      toHtml ":"  
      toHtml (show v)
```

Generando HTML con Shakespeare

```
{-# LANGUAGE QuasiQuotes #-}

import Text.Blaze
import Text.Hamlet

instance ToMarkup [Counter] where
    toMarkup cs = [shamlet|
        <ul>
            $forall c <- cs
                <li>
                    <strong>#{counterName c}
                    : #{counterValue c} |]
```

Bases de datos

Tres enfoques

Una BBDD particular

- postgres-simple
- mysql-simple
- sqlite
- mongoDB
- hedis
- rethinkdb

Basadas en álgebra relacional

- HaskellIDB
- HaSQL

“ORM funcionales”

- **Persistent**
- Groundhog

Primer paso: el esquema

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}  
{-# LANGUAGE GADTs, TypeFamilies #-}  
{-# LANGUAGE MultiParamTypeClasses #-}  
{-# LANGUAGE GeneralizedNewtypeDeriving #-}  
mkPersist sqlSettings [persistLowerCase |  
  Usuario json  
        nombre String  
        apellidos String  
        NombreCompleto nombre apellidos  
deriving Show  
  Contador json  
        nombre String  
        valor Int  
        usuario Usuarioid  
deriving Show []
```

¿Qué hay en un esquema?

Cada elemento del esquema es una *entidad*, compuesta por:

- Una clave primaria siempre presente
 - Cuyo tipo por defecto se denomina EntidadId
- Un conjunto de *campos*: nombreCampo TipoCampo
- Un conjunto de *criterios de unicidad*:
NombreCriterio campo1 campo2 ...

Mirando debajo de la manta: -ddump-splices

```
data Contador
  = Contador { contadorNombre :: !String,
    contadorValor :: !Int,
    contadorUsuario :: !(Key Usuario) }
  deriving (Show, FromJSON, ToJSON)

type ContadorId = Key Contador

instance PersistEntity Contador where
  data Unique Contador
  newtype Key Contador
    = ContadorKey { unContadorKey :: BackendKey SqlBackend }
  data EntityField Contador where
    ContadorId      :: EntityField Contador (Key Contador)
    ContadorNombre :: EntityField Contador String
    ContadorValor   :: EntityField Contador Int
    ContadorUsuario :: EntityField Contador (Key Usuario)
```

Ejecutando una transacción

- Las transacciones se definen en la mónada *SqlPersistT*
 - Una transacción es independiente del *back-end*
- La ejecución de una consulta depende de la BBDD
 - Diferentes librerías como *persistent-sqlite*
 - Normalmente es posible usar un *pool* de conexiones

```
servirListAPI :: ConnectionPool → Server ListAPI
```

```
servirListAPI pool_ = liftIO $
```

```
flip runSqlPersistMPool pool $ do
```

```
-- Aquí va la transacción SQL
```

```
main :: IO ()
```

```
main = runNoLoggingT $
```

```
withSqlitePool "contadores.db" 10 $ λpool →
```

```
NoLoggingT $ run 8081 (servidor pool)
```

Inserción

Consulta por identificador

Consulta por datos

Vamos a ir acabando...

Conclusión

Para saber más

¡Mil gracias!
