Strongly Typed Crystals

Programación Funcional Avanzada

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad Simón Bolívar

UNIVERSIDAD SIMÓN BOLÍVAR

Modelando tipos estructurados

Tipos producto

- Una posibilidad para modelar nombres y apellidos

```
data Nombre = Nombre {
  nombre   :: String,
  apellido :: String
} deriving (Show,Eq)
```

- El compilador genera automáticamente

```
nombre   :: Nombre -> String
apellido :: Nombre -> String
```

Parecen «métodos de acceso» pero no son más que funciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Modelando tipos estructurados

Tipos producto

- Aprovechando notación estándar Haskell para registros

```
> let p = Nombre { nombre = "Mister",
                   apellido = "Magoo" }
> let q = p { apellido = "Robot" }
Nombre { nombre = "Mister", apellido = "Robot" }
> q { nombre = "I" }
Nombre { nombre = "I", apellido = "Robot" }
```

- El compilador ayuda un poco – pero muy poco.

No parecen «métodos de acceso»
porque sólo son «azúcar sintáctica».

UNIVERSIDAD SIMÓN BOLÍVAR

Modelando tipos estructurados

¿Y si se anidan?

- Ahora queremos modelar un cliente

```
data Cliente = Cliente {
  nom :: Nombre,
  rif :: String
} deriving (Show,Eq)
```

- El primer campo no puede llamarse `nombre`.
- Pueden componerse las funciones de acceso

```
> :type nombre . nom
nombre . nom :: Cliente -> String
> :type apellido . nom
apellido . nom :: Cliente -> String
```

¿Cómo cambiar el nombre de un cliente?

UNIVERSIDAD SIMÓN BOLÍVAR

Lenses – Referencias funcionales

`Control.Lens`

- Lenses – construir y manipular funciones para tipos complejos.
- Valor de *primera clase* que representa una función, de un tipo complejo hacia alguno de sus constituyentes.
- Bidireccionales – leer o escribir con copia implícita.
- A simple vista parecen «getters» y «setters».

UNIVERSIDAD SIMÓN BOLÍVAR

Lenses para nuestro tipo de datos

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data Nombre = Nombre {
  _nombre   :: String,
  _apellido :: String
} deriving (Show,Eq)

data Cliente = Cliente {
  _nom :: Nombre,
  _rif :: String
} deriving (Show,Eq)

$(makeLenses ''Cliente)
```

- «Underscore» son convención intencional.
- Template Haskell *genera* Haskell mientras compila.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo se usan?

Sirve para consultar campos

- Supongamos que existe la declaración

```
e = Cliente { _nom = Nombre {
                  _nombre   = "Ernesto",
                  _apellido = "Hernandez-Novich"
                },
                _rif = "V100417088" }
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo se usan?

Sirve para consultar campos

- Supongamos que existe la declaración

```
e = Cliente { _nom = Nombre {
                  _nombre   = "Ernesto",
                  _apellido = "Hernandez-Novich"
                },
                _rif = "V100417088" }
```

- Inmediatamente podríamos hacer

```
> view rif e
"V100417088"
> view nom e
Nombre { _nombre = "Ernesto",
         _apellido = "Hernandez" }
```

- `rif` y `nom` son lenses para los campos de `Cliente` – `view` permite aplicar el lens sobre una estructura `Cliente`.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo se usan?

Sirve para modificar campos

- También podríamos hacer

```
> set rif "V-10041708-8" e
Cliente { _nom = Nombre { _nombre = "Ernesto",
                          _apellido = "Hernandez"},
          _rif = "V-10041708-8"}
```

- No hace falta la copia explícita.

`rif` actuará como lector o copiador
según se aplique con `view` o `set`

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo telescopios

Lenses componen

- Si agregamos

```
$(makeLenses ''Nombre)
```

- Ahora podemos componer para leer – no sorprende.

```
> (view nombre . view nom) e
"Ernesto"
```

- Y se puede hacer usando `view` una sola vez

```
> view (nom . nombre) e
"Ernesto"
```

Componen de izquierda a derecha –
para los que necesitan «métodos» en su vida.

Lenses – foco ajustable

- Un lens abstrae «consultar» o «cambiar» en una función única.
- No más notación de registros.
- Componen «a lo programación imperativa»
- Permiten aplicar funciones vía un lens – un `fmap` telescópico.

```
> (nom . nombre) `over` (map toUpper) $ e
Cliente { _nom = Nombre { _nombre = "ERNESTO",
                          _apellido = "Hernandez"},
          _rif = "V100417088"}
```

- `over` recibe un lens y una función pura conveniente – enfoca en profundidad y copia implícitamente.

The Joy of Abstraction!

UNIVERSIDAD SIMÓN BOLÍVAR

Los tipos suelen explicar todo

- Mirando el código de la librería, se encuentra

```
view :: Lens' a b -> a -> b
set  :: Lens' a b -> b -> a -> a
```

  - `view` se enfoca en obtener el valor foco.
  - `set` se enfoca en crear una copia cambiada.

- La composición de Prelude generaliza a

```
(.) :: Lens' a b -> Lens' b c -> Lens' a c
```

  que explica el curioso (¡y agradable!) orden de composición.

- Más aún, la identidad de Prelude generaliza a

```
id :: Lens' a a
```

UNIVERSIDAD SIMÓN BOLÍVAR

Lense Laws

Que son bastante sensatas

Lense Laws

Que son bastante sensatas

1. *Put then Get* – Si cambias algo, será lo que leas inmediatamente

```
view lens (set lens new this) = new
```

Lense Laws

Que son bastante sensatas

1. *Put then Get* – Si cambias algo, será lo que leas inmediatamente

```
view lens (set lens new this) = new
```

2. *Get then Put* – Si cambias usando lo que se lee, no cambia nada

```
set lens (view lens this) this = this
```

UNIVERSIDAD SIMÓN BOLÍVAR

Lense Laws

Que son bastante sensatas

1. *Put then Get* – Si cambias algo, será lo que leas inmediatamente

```
view lens (set lens new this) = new
```

2. *Get then Put* – Si cambias usando lo que se lee, no cambia nada

```
set lens (view lens this) this = this
```

3. *Put then Put* – Cambiar dos veces, deja el último cambio

```
set lens v2 (set lens v1 this) = set lens v2 this
```

Sistema de Tipos no puede verificarlas
La librería genera código que lo cumple.

UNIVERSIDAD SIMÓN BOLÍVAR

Smooth Operators

Para leer con aire imperativo

- Versión infija "flipped" de `view`

```
(^.) :: a -> Lens' a b -> b
```

- Nos permite escribir

```
> e ^. nom . nombre
"Ernesto"
> e ^. nom . apellido
"Hernandez"
> e ^. rif
"V10041708"
```

UNIVERSIDAD SIMÓN BOLÍVAR

Smooth Operators

Para escribir con aire imperativo

- Versión infija "flipped" de `set`

```
(.~) :: Lens' a b -> b -> a -> a
```

- Nos permite escribir

```
> (nom . nombre) .~ "Santiago" $ e
Cliente { _nom = Nombre { _nombre = "Santiago",
                          _apellido = "Hernandez"},
          _rif = "V100417088"}
```

UNIVERSIDAD SIMÓN BOLÍVAR

Enfocándonos en el estado mutable

- Tipos complejos frecuentes para estado mutable en `State`.
- Combinadores especializados

```
(.=)   :: MonadState a m => Lens' a b -> b -> m ()
use    :: MonadState a m => Lens' a b -> m b
(+=)   :: (MonadState s m, Num a) => Lens' s a -> a
                                  -> m ()
(||=) :: MonadState s m => Lens' s Bool -> Bool
                        -> m ()
```

No más get ni put

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si los tipos son polimórficos?

Por aquello de la generalidad...

- Tuplas `(a,b)` – el tipo producto polimórfico más sencillo.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si los tipos son polimórficos?

Por aquello de la generalidad...

- Tuplas `(a,b)` – el tipo producto polimórfico más sencillo.
- Un lens por cada posición – definidos en `Control.Lens`.

```
_1 :: Lens' (a,b) -> a
_2 :: Lens' (a,b) -> b
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si los tipos son polimórficos?

Por aquello de la generalidad...

- Tuplas `(a,b)` – el tipo producto polimórfico más sencillo.
- Un lens por cada posición – definidos en `Control.Lens`.

```
_1 :: Lens' (a,b) -> a
_2 :: Lens' (a,b) -> b
```

- Funcionan como «getters»...

```
> view (_2 . _1) (42,("wat",True))
"wat"
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si los tipos son polimórficos?

Por aquello de la generalidad...

- Tuplas `(a,b)` – el tipo producto polimórfico más sencillo.
- Un lens por cada posición – definidos en `Control.Lens`.

```
_1 :: Lens' (a,b) -> a
_2 :: Lens' (a,b) -> b
```

- Funcionan como «getters»...

```
> view (_2 . _1) (42,("wat",True))
"wat"
```

- ...y como «setters» cambiando el tipo final

```
> set (_2 . _1) 69.0 (42,("wat",True))
(42,(69.0,True))
```

Librería es capaz de generar lenses polimórficos – sistema de tipos verifica consistencia.

UNIVERSIDAD SIMÓN BOLÍVAR

Prismas – Selectores funcionales

Dualidad óptica – pun intended

- Prismas – enfocarse en una alternativa de un tipo suma
- Escoger una alternativa de manera segura.

```
preview :: Prism' s a -> s -> Maybe a
```

- Regresar al punto anterior de selección.

```
review :: Prism' s a -> a -> s
```

`Control.Lens` incluye varios prismas

UNIVERSIDAD SIMÓN BOLÍVAR

Prismas para Either – _Left y _Right

- Para escoger una alternativa de manera segura

```
> preview _Left (Left 42)
Just 42
> preview _Right (Left 42)
Nothing
```

Prismas para Either – _Left y _Right

- Para escoger una alternativa de manera segura

```
> preview _Left (Left 42)
Just 42
> preview _Right (Left 42)
Nothing
```

- Para regresar al nivel anterior

```
> review _Left 42
Left 42
> review _Right 42
Right 42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Prismas para Either – _Left y _Right

- Para escoger una alternativa de manera segura

```
> preview _Left (Left 42)
Just 42
> preview _Right (Left 42)
Nothing
```

- Para regresar al nivel anterior

```
> review _Left 42
Left 42
> review _Right 42
Right 42
```

- El operador infijo ^? es muy conveniente

```
> (Right 42) ^? _Right
Just 42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Prisma para [a] – _Cons

- Para escoger una alternativa de manera segura

```
> preview _Cons []
Nothing
> preview _Cons [42,23,69]
Just (42,[23,69])
```

UNIVERSIDAD SIMÓN BOLÍVAR

Prisma para [a] – _Cons

- Para escoger una alternativa de manera segura

```
> preview _Cons []
Nothing
> preview _Cons [42,23,69]
Just (42,[23,69])
```

- Para regresar al nivel anterior

```
> review _Cons (42,[23,69])
[42,23,69]
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y funciona con mis tipos de datos?

. . . polimórficos, de una vez.

- En lugar de lenses, pedimos prismas.

```
data Meta a = Foo a | Bar a

$(makePrisms ''Meta)
```

- La librería agregará los underscores necesarios

```
> preview _Bar (Bar 42)
Just 42
> preview _Foo (Bar 42)
Nothing
> review _Foo 42
Foo 42
```

Más generales que los «Zippers»

UNIVERSIDAD SIMÓN BOLÍVAR

Procesar JSON

La lingua franca de los servicios web

- Formato JSON para intercambio de datos
  - Basado en texto simple.
  - Estructura legible para el humano.
  - Subconjunto propio de JavaScript para serialización.
- Objetos construidos como listas nombre/valor.
  - Nombres – cadenas alfanuméricas.
  - Valores – `null`, cadenas, números, booleanos, arreglos unidimensionales u objetos anidados.
- Siempre hace falta un reconocedor.

Mejor que XML...

UNIVERSIDAD SIMÓN BOLÍVAR

JSON en Haskell

Aprovechando el sistema de tipos

- Haskell incluye reconocedores y emisores de JSON – como casi cualquier otro lenguaje.
- `Data.Aeson` – altamente optimizada y fácil de usar.
- `Data.Aeson.Encode.Pretty` – mostrar JSON "bonito"

UNIVERSIDAD SIMÓN BOLÍVAR

Una encuesta sobre pizza

. . . de una aplicación web, de un startup, blah

```
data Person = Person {
    firstName  :: !Text,
    lastName   :: !Text,
    age        :: !Int,
    likesPizza :: !Bool
} deriving (Show)
```

- La aplicación manejará los datos internamente en este formato.
- Los desarrolladores del *front-end* necesitan JSON.

UNIVERSIDAD SIMÓN BOLÍVAR

El API fundamental

`Data.Aeson`

- Codificar JSON hacia un `ByteString`

```
encode :: ToJSON a => a -> ByteString
```

- Decodificar entrada JSON desde un `ByteString`

```
decode :: FromJSON a => ByteString -> Maybe a
```

- Decodificador con descripción de errores

```
eitherDecode :: FromJSON a
             => ByteString -> Either String a
```

- Variantes estrictas en el resultado

```
decode'       :: FromJSON a
              => ByteString -> Maybe a
eitherDecode' :: FromJSON a
              => ByteString -> Either String a
```

UNIVERSIDAD SIMÓN BOLÍVAR

JSON desde y hacia archivos

Intercambio de datos local

```
[
   { "firstName"   : "John"
   , "lastName"    : "Doe"
   , "age"         :  24
   , "likesPizza"  :  true
   }
...
]
```

- No tiene por qué estar indentado.
- Podría ser *gigantesco* – lazy `ByteString` FTW!
- Nos encantaría terminar con `[Person]`.

UNIVERSIDAD SIMÓN BOLÍVAR

JSON desde archivos

La magia de GHC

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}
import Data.Text
import GHC.Generics

data Person = Person {
    firstName  :: !Text,
    lastName   :: !Text,
    age        :: Int,
    likesPizza :: Bool
} deriving (Show,Generic)

instance FromJSON Person
instance ToJSON Person
```

- ¿ByteString, Text o String? – OverLoadedStrings
- GHC genera instancias FromJSON y ToJSON automáticamente.

UNIVERSIDAD SIMÓN BOLÍVAR

JSON desde archivos

Pruebas con valores inmutables

```
> eitherDecode $
    fromString goodjson :: Either String [Person]
Right [Person {firstName = "John", ... },
       Person {firstName = "Rose", ... }]
> eitherDecode $
    fromString badjson :: Either String [Person]
Left "when expecting a Bool, encountered Number instead"
```

- Decodificación requiere `Text` o `ByteString` – por eso `fromString`
- Resultado o error de sintaxis explicativo.

UNIVERSIDAD SIMÓN BOLÍVAR

JSON desde archivos

¿Y desde el archivo?

```
main = do
  d <- (eitherDecode <$> B.readFile "pizza.json")
       :: IO (Either String [Person]
  case d of
    Left err -> putStrLn err
    Right ps -> print ps
```

- En este caso es necesario anotar el tipo deseado –
  en otros programas podría no hacer falta según contexto.
- Aprovechar `Control.Applicative`

UNIVERSIDAD SIMÓN BOLÍVAR

JSON hacia archivos

Codificar datos Haskell en JSON

```
import qualified Data.ByteString.Lazy      as B
import qualified Data.ByteString.Lazy.Char8 as B8

person = Person {
  firstName = "Ernesto",
  lastName = "áHernndez-Novich",
  age = 45,
  likesPizza = True
}

> B8.putStrLn $ encode person
{"lastName":"áHernndez-Novich","age":45,
 "firstName":"Ernesto","likesPizza":true}
```

- Tanto `Person` como `[Person]` son codificables.
- Emitir `ByteSring` con la codificación adecuada.

UNIVERSIDAD SIMÓN BOLÍVAR

¡Ay, pero que feo!

A los programas sólo les importa la belleza interna

```
import Data.Aeson.Encode.Pretty

B8.putStrLn $ encodePretty person
{
    "age": 45,
    "firstName": "Ernesto",
    "lastName": "Hernandez-Novich",
    "likesPizza": true
}
```

- Indentado y en orden alfabético.
- `encodePretty'` es configurable – indentación y orden.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si está en un servicio web?

Can I haz download?

```
import Network.HTTP.Conduit (simpleHttp)

getJSON :: IO B.ByteString
getJSON = simpleHttp "http://localhost/~emhn/pizza.json"

main = do
  d <- (eitherDecode <$> getJSON)
       :: IO (Either String [Person])
  ...
```

- Conduit modela *streams* perezosos – archivos, HTTP...

UNIVERSIDAD SIMÓN BOLÍVAR

Más flexibilidad

Data.Aeson

- Instancias `FromJSON` y `ToJSON` para muchos tipos Haskell.

```
> decode "[1,2,3]" :: Maybe [Int]
Just [1,2,3]
> decode "{\"foo\":1,\"bar\":2}"
  :: Maybe (Map String Int)
Just (fromList [("bar",2),("foo",1)])
```

UNIVERSIDAD SIMÓN BOLÍVAR

La implantación

Parsec y librerías especializadas

- Objeto JSON es mapa de claves a valores – `Data.HashMap`

```
type Object = HashMap Text Value
```

- Arreglo (lista) JSON es un vector de valores – `Data.Vector`

```
type Array = Vector Value
```

- Un valor JSON puede ser

```
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null
```

- Reconocedor escrito con `AttoParsec` y `Blaze`

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué tiene que ver con Lenses?

- Entre la lectura y la escritura, suele haber modificaciones.
- En general, JSON tiene estructuras anidadas.
- `Data.Aeson.Lens` – colección de Lenses, Prisms y Traversals para procesar JSON.
  - `nth` – el n—ésimo elemento de una lista, desde cero.
  - `key` – la clave en un objeto.
  - `_Object`, `_Array`, `_String`, `_Number` – alternativas JSON.

UNIVERSIDAD SIMÓN BOLÍVAR

Decodificar y analizar al vuelo

- Supongamos que `j` tiene el siguiente `Text`

```
[{"firstName":"John","lastName":"Doe","age":24,...
 {"firstName":"Rose","lastName":"Red","age":39,...
```

Decodificar y analizar al vuelo

- Supongamos que `j` tiene el siguiente `Text`

```
[{"firstName":"John","lastName":"Doe","age":24,...
 {"firstName":"Rose","lastName":"Red","age":39,...
```

- Entonces podemos navegar la estructura a gusto

```
> j ^? nth 1
Just (Object (fromList [("lastName",String "Red"),...
> j ^? nth 1 . key (T.pack "age")
Just (Number 39.0)
> j ^? nth 1 . key (T.pack "age") . _Integer
Just 39
> j ^? nth 1 . key (T.pack "noexiste")
Nothing
```

UNIVERSIDAD SIMÓN BOLÍVAR

Decodificar y analizar al vuelo

- Supongamos que `j` tiene el siguiente `Text`

```
[{"firstName":"John","lastName":"Doe","age":24,...
 {"firstName":"Rose","lastName":"Red","age":39,...
```

- Entonces podemos navegar la estructura a gusto

```
> j ^? nth 1
Just (Object (fromList [("lastName",String "Red"),...
> j ^? nth 1 . key (T.pack "age")
Just (Number 39.0)
> j ^? nth 1 . key (T.pack "age") . _Integer
Just 39
> j ^? nth 1 . key (T.pack "noexiste")
Nothing
```

- Y modificarla

```
> j & nth 1 . key (T.pack "age") . _Integer .~ 42
"[ (...) ,{\"lastName\":\"Red\",\"age\":42, ...
```

Quiero saber más...

- Lenses and Functional References – Haskell WikiBook
- Lens Over Tea – Lenses, Traversals and some implementation details
- Documentación de Control.Lens
- RFC-4627 – The application/json Media Type for JSON
- Documentación de `Data.Aeson`

UNIVERSIDAD SIMÓN BOLÍVAR