Universidad de la República, Uruguay

Proyecto de Grado

Desarrollo de DSLs en lenguajes con tipos dependientes

Gonzalo Waszczuk

Supervisado por
Marcos Viera
Alberto Pardo

Resumen

Pendiente: Resumen

Índice general

Capítulo 1

Introducción

Los records extensibles son una herramienta muy útil en la programación. Su necesidad ocurre de un problema que tienen lenguajes estáticos en relación a records: ¿Cómo puedo modificar la estructura de un record ya definido?

Sin embargo, en los lenguajes de programación fuertemente tipados modernos no existe una forma canónica de definir y manipular records extensibles. Algunos lenguajes ni si quiera permiten definirlos.

En este trabajo se presenta una manera de definir records extensibles y poder trabajar con ellos de forma simple, utilizando un lenguaje de programación con *tipos dependientes* llamado Idris.

1.1. Records Extensibles

En varios lenguajes de programación con un sistema de tipos estáticos, es posible definir una estructura estática llamada 'record'. Un record permite agrupar varios valores en un único conjunto, asociando una etiqueta o nombre a cada uno de esos valores. Un ejemplo de un record en Haskell sería el siguiente

```
data Persona = Persona { edad :: Int, nombre :: String}
```

Una desventaja que tienen los records es que una vez definidos, no es posible modificar su estructura de forma dinámica. Tomando el ejemplo anterior, si uno quisiera tener un nuevo record con los mismos campos de `Persona` pero con un nuevo campo adicional, como `apellido`, entonces uno solo podría definirlo de una de estas dos formas:

- Definir un nuevo record con esos 3 campos
- Crear un nuevo record que contenga el campo `apellido` y contenga un campo de tipo `Persona`

En ninguno de ambos enfoques es posible obtener el nuevo record de manera dinámica. Es decir, siempre es necesario definir un nuevo tipo, indicando que es un record e indicando sus campos.

Los records extensibles intentan resolver ese problema. Si `Persona` fuera un record extensible, entonces definir un nuevo record idéntico a él, pero con un campo adicional, sería tan fácil como tomar un record de `Persona` existente, y simplemente

agregarle el nuevo campo `apellido` con su valor correspondiente. A continuación se muestra un ejemplo de record extensible, con una sintaxis hipotética similar a Haskell

```
p :: Persona
p = Persona {edad=20, nombre="Juan"}

pExtensible :: Persona + {apellido :: String}
pExtensible = p + {apellido="Sanchez"}
```

El tipo del nuevo record debería reflejar el hecho de que es una copia de `Persona` pero con el campo adicional utilizado. Uno debería poder, por ejemplo, acceder al campo `apellido` del nuevo record como uno lo haría con cualquier otro record arbitrario.

Para poder utilizar un record extensible, es necesario poder codificar toda la información del record. Esta información incluye las etiquetas que acepta el record, y el tipo de los valores asociados a cada una de esas etiquetas. Si en un lenguaje de programación con tipado estático no existe soporte del lenguaje para manejar estos records, entonces es necesario mantener esa información en el *tipo* del record. Los *tipos dependientes* permiten llevar a cabo esto.

1.2. Tipos Dependientes

Con tipos dependientes, se permite que un tipo dependa de valores y no solo de tipos. Un ejemplo es el de un tipo que representa un natural con cota superior:

```
Fin : Nat -> Type
```

Este tipo es parametrizado por un natural, el cual indica el valor de la cota superior. Por lo tanto un valor de tipo `Fin 10` solamente puede ser un natural menor a 10, mientras que un valor de tipo `Fin 4` solamente puede ser un natural menor a 4.

Otro ejemplo es el de un vector o lista donde su tamano esta indicado en su tipo:

```
data Vect : Nat -> Type -> Type where
  [] : Vect 0 A
  (::) : (x : A) -> (xs : Vect n A) -> Vect (n + 1) A
```

Un valor del tipo `Vect 1 String` equivale a una lista de 1 string, mientras que un valor del tipo `Vect 10 String` equivale a una lista de 10 strings.

Tener un valor en el tipo permite poder restringir el uso de funciones a determinados tipos que tengan un valor específico. Por ejemplo se tiene la siguiente definición de función:

```
index : Fin n -> Vect n a -> a
```

Esta función obtiene un valor de un vector dado su índice. Sin embargo, si el vector tiene largo `n`, esta función fuerza a que el índice pasado por parámetro sea menor a `n` para que `Fin n` pueda ser construido.

En relación a records extensibles, los tipos dependientes hacen posible que dentro del tipo del record puedan existir valores para poder ser manipulados. Posibles valores

podrían ser la lista de etiquetas del record. Como esta información se encuentra en el tipo del record, es posible definir tipos y funciones que accedan a ella y la manipulen para poder definir todas las funcionalidades de records extensibles.

1.3. Idris

En este trabajo se decidió utilizar el lenguaje de programación *Idris* para llevar a cabo la investigación de tipos dependientes aplicados a records extensibles.

Idris es un lenguaje de programación funcional con tipos dependientes, con sintaxis similar a Haskell.

A continuación se muestra un ejemplo de código escrito en Idris:

```
length : (xs : Vect n a) -> Nat
length [] = 0
length (x::xs) = 1 + length xs
```

El código descrito en la sección anterior también está en Idris.

Otro lenguaje que cumple con los requisitos es *Agda*. *Agda* es otro lenguaje funcional con tipos dependientes. Sin embargo, se decidió utilizar Idris para investigar las funcionalidades de este nuevo lenguaje. Por este motivo, todas las conclusiones obtenidas en este informe pueden ser aplicadas a *Agda* también.

1.3.1. Tipos Decidibles

Una noción que surgió constantemente en el trabajo presente es el de tipos decidibles. En Idris la decidibilidad de un tipo se representa con el siguiente predicado:

```
data Dec : Type -> Type where
  Yes : (prf : prop) -> Dec prop
  No  : (contra : Not prop) -> Dec prop
```

Un tipo es decidible si se puede construir un valor de si mismo, o se puede construir un valor de su contradicción. Si se tiene `Dec P`, entonces significa que o bien existe un valor `s : P` o existe un valor `n : Not P`. Cabe notar que `Not P` es equivalente a `P ->Void`, representando la contradicción de `P`.

Poder obtener tipos decidibles es importante cuando se tienen tipos que funcionan como predicados y se necesita saber si ese predicado se cumple o no. Solo basta tener un `Dec P` para poder realizar un análisis de casos, uno cuando `P` es verdadero y otro cuando no.

Otra funcionalidad importante es la de poder realizar igualdad de valores:

```
class DecEq t where
  total decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
```

`DecEq t` indica que, siempre que se tienen dos elementos `x1, x2 : t`, siempre es posible tener una prueba de que son iguales o una prueba de que son distintos. La función `decEq` es importante cuando se quiere realizar un análisis de casos sobre la igualdad de dos elementos, un caso donde son iguales y otro caso donde se tiene una prueba de que no lo son.

Los tipos decidibles y las funciones que permiten obtener valores del estilo `Dec P` son muy importantes al momento de probar teoremas y manipular predicados.

Capítulo 2

Estado del arte

2.1. Records Extensibles como parte del lenguaje

Algunos lenguaje de programación funcionales permiten el manejo de records extensibles como funcionalidad del lenguaje.

Uno de ellos es *Elm*. Uno de los ejemplos que se muestra en su documentación ([2]) es el siguiente:

```
type alias Positioned a =
  { a | x : Float, y : Float }

type alias Named a =
  { a | name : String }

type alias Moving a =
  { a | velocity : Float, angle : Float }

lady : Named { age:Int }
lady =
  { name = "Lois Lane"
  , age = 31
  }

dude : Named (Moving (Positioned {}))
dude =
  { x = 0
  , y = 0
  , name = "Clark Kent"
  , velocity = 42
  , angle = degrees 30
  }
```

Elm permite definir tipos que equivalen a records, pero agregándole campos adicionales, como es el caso de los tipos descritos arriba. Este tipo de record extensible hace uso de *row polymorphism*. Básicamente, al definir un record, éste se hace polimórfico sobre el resto de los campos. Es decir, se puede definir un record

que traiga como mínimo unos determinados campos, pero el resto de éstos puede variar. En el ejemplo de arriba, el record `lady` es definido extendiendo `Named` con otro campo adicional, sin necesidad de definir un tipo nuevo.

Una desventaja es que por el poco uso de records extensibles en la versión 0.16 se decidió eliminar la funcionalidad de agregar y eliminar campos a records de forma dinámica [1].

Otro lenguaje con esta particularidad es *Purescript*. A continuación se muestra uno de los ejemplos de su documentación ([10]):

```
fullname :: forall t. { firstName :: String,
  lastName :: String | t } -> String
fullName person = person.firstName ++ " " ++ person.lastName
```

Purescript permite definir records con determinados campos, y luego definir funciones que solo actúan sobre los campos necesarios. Utiliza *row polymorphism* al igual que Elm.

Ambos lenguajes basan su implementación de records extensibles, aunque sea parcialmente, en el paper [7].

También existen otras propuestas de sistemas de tipos con soporte para records extensibles. Algunas de ellas son [8], [4], y [3].

Todas estas propuestas tienen en común la necesidad de extender el lenguaje para poder soportar records extensibles. Esto tiene el problema de que los records extensibles no tienen el soporte *first-class* de otros tipos del lenguaje. Esta falta de soporte limita la expresividad de tales records, ya que cualquier manipulación de ellos debe ser proporcionada por el lenguaje, cuando podrían ser proporcionadas por librerías de usuario.

Para poder definir records extensibles con la propiedad de *first-class citizens*, generalmente se utilizan listas heterogéneas para hacerlo.

2.2. Listas heterogéneas

El concepto de listas heterogéneas (o *HList*) surge en oposición al tipo de listas más utilizado en la programación con lenguajes tipados: listas homogéneas. Las listas homogéneas son las más comunes de utilizar en estos lenguajes, ya que son listas que pueden contener elementos de un solo tipo. Estos tipos de listas existen en todos o casi todos los lenguajes de programación que aceptan tipos parametrizables o genéricos, sea Java, C#, Haskell y otros. Ejemplos comunes son `List<int>` en Java, o `[String]` en Haskell.

Las listas heterogéneas, sin embargo, permiten almacenar elementos de cualquier tipo arbitrario. Estos tipos pueden o no tener una relación entre ellos, aunque en general se llama 'listas heterogénea' a la estructura de datos que no impone ninguna relación entre los tipos de sus elementos.

En lenguajes dinámicamente tipados (como lenguajes basados en LISP) este tipo de listas es fácil de construir, ya que no hay una imposición por parte del typechecker sobre qué tipo de elementos se pueden insertar o pueden existir en esta lista. El siguiente es un ejemplo de lista heterogénea en un lenguaje LISP como Clojure o Scheme.

```
'(1 0.2 "Text")
```

Sin embargo, en lenguajes tipados este tipo de lista es más dificil de construir. Para determinados lenguajes no es posible incluir la mayor cantidad de información posible en tales listas para poder trabajar con sus elementos, ya que sus sistemas de tipos no lo permiten. Esto se debe a que tales listas pueden tener elementos con tipos arbitrarios, por lo tanto los sistemas de tipos de estos lenguajes necesitan una forma de manejar tal conjunto arbitrario de tipos.

En sistemas de tipos más avanzados, es posible incluir la mayor cantidad posible de información de tales tipos arbitrarios en el mismo tipo de la lista heterogénea. A continuación se describe la forma de crear listas heterogéneas en Haskell, y luego la forma de crearlas en Idris, utilizando el poder de tipos dependientes de este lenguaje para llevarlo a cabo.

2.2.1. HList en Haskell

Para definir listas heterogéneas en Haskell nos basaremos en la propuesta presentada en [6], e implementada en el paquete *hlist*, encontrado en Hackage [5].

Esta librería define HList de la siguiente forma:

```
data family HList (l::[*])

data instance HList '[] = HNil
data instance HList (x ': xs) = x `HCons` HList xs
```

Se utilizan *data families* para poder crear una familia de tipos `HList`, utilizando la estructura recursiva de las listas, definiendo un caso cuando una lista está vacía y otro caso cuando se quiere agregar un elemento a su cabeza. Esta definición representa una secuencia de tipos separados por `HCons` y terminados por `HNil`. Esto permite, por ejemplo, construir el siguiente valor

```
10 `HCons` ('Text' `HCons` HNil) :: HList '[Int, String]
```

Esta definición hace uso de una forma de tipos dependientes, al utilizar una lista de tipos como parámetro de `HList`. La estructura recursiva de `HList` garantiza que es posible construir elementos de este tipo a la misma vez que se van construyendo elementos de la lista de tipos que se encuentra parametrizada en `HList`.

Para poder definir funciones sobre este tipo de listas, es necesario utilizar *type families*, como muestra el siguiente ejemplo

```
type family HLength (x :: [k]) :: HNat
type instance HLength '[] = HZero
type instance HLength (x ': xs) = HSucc (HLength xs)
```

2.2.2. HList en Idris

Uno de los objetivos de la investigación de listas heterogéneas en un lenguaje como Idris es poder utilizar el poder de su sistema de tipos. Esto evite que uno tenga que recurrir a este tipo de construcciones para poder manejar listas heterogéneas. Una de las metas es poder programar utilizando listas heterogéneas con la misma facilidad que uno programa con listas homogéneas. Esto último no ocurre en el caso

de HList en Haskell, ya que el tipo de HList es definido con data families, y funciones sobre HList son definidas con type families. Esta definición se contrasta con las listas homogéneas, cuyo tipo se define como un tipo común, y funciones sobre tales se definen como funciones normales también.

A diferencia de Haskell, el lenguaje de programación Idris maneja tipos dependientes completos. Esto significa que cualquier tipo puede estar parametrizado por un valor, y este tipo es un *first-class citizen* que puede ser utilizado como cualquier otro tipo del lenguaje. Esto significa que para Idris no hay diferencia en el trato de un tipo simple como `String` y en el de un tipo complejo con tipos dependientes como `Vect 2 Nat`.

La definición de HList utilizada en Idris es la siguiente:

```
data HList : List Type -> Type where
  Nil : HList []
  (::) : t -> HList ts -> HList (t :: ts)
```

El tipo `HList` se define como una función de tipo que toma una lista de tipos y retorna un tipo. Éste se construye definiendo una lista vacía que no tiene tipos, o definiendo un operador de *cons* que tome un valor, una lista previa, y agregue ese valor a la lista. En el caso de *cons*, no solo agrega el valor a la lista, sino que agrega el tipo de tal valor a la lista de tipos que mantiene `HList` en su tipo.

Esta definición de HList permite construir listas heterogéneas con relativa facilidad, como por ejemplo

```
23 :: "Hello World" :: [1,2,3] :
    HList [Nat, String, [Nat]]
```

Otra de las ventajas de esta definición es el hecho de que los tipos de los elementos de la lista están presentes en el tipo mismo. Esto permite que uno pueda siempre recuperar uno de estos tipos si uno quiere utilizarlo luego.

A su vez, a diferencia de Haskell, la posición de *first-class citizens* de los tipos dependientes en Idris permite definir funciones con pattern matching sobre HList.

```
hLength : HList ls -> Nat
hLength Nil = 0
hLength (x :: xs) = 1 + (hLength xs)
```

Aquí surge una diferencia muy importante con la implementación de HList de Haskell, ya que en Idris `hLength` puede ser utilizada como cualquier otra función, y en especial opera sobre *valores*, mientras que en Haskell `HLength` solo puede ser utilizada como type family, y solo opera sobre *tipos*. Esto permite que el uso de HList en Idris no sea distinto del uso de listas comunes (`List`), y por lo tanto puedan tener el mismo trato de ellas para los creadores de librerías y los usuarios de tales, disminuyendo la dificultad de su uso para resolver problemas.

2.3. Records Extensibles en Haskell

Algunas de las propuestas de records extensibles en Haskell, utilizando listas heterogéneas, son [9] y [6].

Para este trabajo nos basaremos en la propuesta de [6] que ya utilizamos para definir listas heterogéneas anteriormente.

En este enfoque, se tiene un tipo `Tagged s b` que se define de la siguiente forma:

```
data Tagged s b = Tagged b
```

Este tipo permite tener un *phantom type* en el tipo `s`. Esto significa que un valor de tipo `Tagged s b` va a contener solamente un valor de tipo `b`, pero en tiempo de compilación se va a tener el tipo `s` para manipular.

Luego esta librería define la siguiente clase

```
class (HLabelSet (LabelsOf ps), HAllTaggedLV ps) =>
  HRLabelSet (ps :: [*])
```

Dado un tipo `ps` que se corresponde a un `HList` que contiene solamente valores del tipo `Tagged s b`, esta clase indica que esa lista heterogénea se corresponde a un conjunto de etiquetas. El tipo `LabelsOf ps` retorna los *phantom types* de un valor tagueado, considerados como 'etiquetas'. La clase `HLabelSet (LabelsOf ps)` indica que esas etiquetas son un conjunto, es decir, no tienen valores repeditos. Finalmente la clase `HAllTaggedLV ps` indica que `ps` es compuesto solamente por valores del tipo `Tagged s b`. `HLabelSet` fuerza a que cada tipo `s` pueda igualarse con los demás, pero el tipo `b` es arbitrario y puede ser cualquiera.

Dada esta clase, esta libería define un record de esta forma:

```
newtype Record (r :: [*]) = Record (HList r)

mkRecord :: HRLabelSet r => HList r -> Record r
mkRecord = Record
```

Un record es simplemente una lista heterogénea de valores de tipo `Tagged s b`, tal que `s` se puede chequear por igualdad, y tal que todos esos tipos `s` sean distintos y formen un conjunto de etiquetas.

Todas las funciones sobre records extensibles se definen utilizando typeclasses para realizar la computación en los tipos. Un ejemplo de ello es la función que obtiene un elemento de un record dada su etiqueta:

```
class HasField (l::k) r v | l r -> v where
  hLookupByLabel:: Label l -> r -> v

(.!.) :: (HasField l r v) => r -> Label l -> v
r .!. l =  hLookupByLabel l r
```

Capítulo 3

Records Extensibles en Idris

3.1. Introducción

PENDIENTE

3.2. HList actualizado

En este trabajo se decidió implementar las listas heterogéneas de una forma distinta a la que utiliza Idris en general. Las listas heterogéneas de Idris permiten incluir cualquier tipo arbitrario, pero eso no es suficiente al momento de poder implementar records extensibles. Para implementar records extensibles, no solo es necesario poder tener tipos arbitrarios en tal lista, sino que es necesario asociar una etiqueta a cada uno de esos tipos.

Una posible implementación es la de `Record` de Haskell definida en el capítulo anterior. En ésta se define un tipo `Tagged s b` que contenga un *phantom type*, de tal forma que luego se utiliza una lista de esos tipos. Sin embargo se decidió no utilizar esta implementación, ya que necesita definir tipos auxiliarse (como `Tagged`), y a su vez necesita definir predicados específicos para esta implementación, como el predicado que indica que la lista heterogénea está compuesta solamente por valores del tipo `Tagged`.

Se decidió utilizar una solución más simple, en donde se reimplementa `HList` para que contenga la etiqueta junto al tipo

```
LabelList : Type -> Type
LabelList lty = List (lty, Type)

data HList : LabelList lty -> Type where
  Nil : HList []
  (::) : {lbl : lty} -> (val : t) -> HList ts ->
    HList ((lbl,t) :: ts)
```

El tipo `LabelList` representa una lista de tipos, junto a una etiqueta para cada uno de esos tipos. Las etiquetas pueden ser de cualquier tipo, pero generalmente se utiliza `String` para poder nombrarlas mejor. Un ejemplo de tal lista sería `[("1",`

`Nat),("2", String)] : LabelList String`. El resto de la implementación es idéntica a la de `HList` de Idris, con la diferencia de que se debe pasar no solo el valor sino la etiqueta de su campo también.

3.3. Records Extensibles

PENDIENTE

3.4. Alternativas de HList en Idris

El sistema de tipos de Idris permite definir tipos de muchas maneras, por lo cual en su momento se investigaron otras formas distintas de implementar HList. A continuación se describen tales formas, indicando por qué no se optó por cada una ellas.

3.4.1. Dinámico

```
data HValue : Type where
    HVal: {A : Type} -> (x : A) -> HValue

HList : Type
HList = List HValue
```

Este tipo se asemeja al tipo `Dynamic` utilizado a veces en Haskell, o a `Object` en Java/C#. Esta HList mantiene valores de tipos arbitrarios dentro de ella, pero no proporciona ninguna información de ellos en su tipo. Cada valor es simplemente reconocido como `HValue`, y no es posible conocer su tipo u operar con él de ninguna forma. Solo es posible insertar elementos, y manipularlos en la lista (eliminarlos, reordenarlos, etc).

Este enfoque se asemeja al uso de `List<Object>` de Java/C# que fue usado incluso antes de que estos lenguajes tuvieran tipos parametrizables, pero sin la funcionalidad de poder realizar *down-casting* (castear un elemento de una superclase a una subclase).

No se utilizó este enfoque ya que al no poder obtenerse la información del tipo de `HValue` es imposible poder verificar que un record contenga campos con etiquetas y que éstos no estén repetidos, al igual que es imposible poder trabajar con tal record luego de construido.

Un ejemplo de su uso es

```
[HVal (1,2), HVal "Hello", HVal 42] : HList
```

3.4.2. Existenciales

```
data HList : Type where
    Nil : HList
    (::) : {A : Type} -> (x : A) -> HList -> HList
```

Este enfoque se asemeja al uso de *tipos existenciales* utilizado en Haskell. Básicamente el tipo `HList` se define como un tipo simple sin parámetros, pero sus constructores permiten utilizar valores de cualquier tipo. Esta definición es muy similar a la que utiliza tipos dinámicos, y tiene las mismas desventajas. Luego de creada una HList de esta forma, no es posible obtener ninguna información de los tipos de los valores que contiene, por lo que es imposible poder trabajar con tales valores. Por ese motivo tampoco fue utilizado.

Un ejemplo de su uso es

```
[1,"2"] : HList
```

3.4.3. Estructurado

```
using (x : Type, P : x -> Type)
    data HList : (P : x -> Type) ->  Type where
        Nil : HList P
        (::) : {head : x} -> P head -> HList P ->
          HList P
```

Esta definición es un punto medio (en términos de poder) entre la definición utilizada en este trabajo y las demás definiciones descritas en las secciones anteriores.

Esta HList es parametrizada sobre un constructor de tipos. Es decir, toma como parámetro una función que toma un tipo y construye otro tipo a partir de éste. Esta definición permite imponer una estructura en común a todos los elementos de la lista, forzando que cada uno de ellos haya sido construido con tal constructor de tipo, sin importar el tipo base utilizado. La desventaja es que no es posible conocer nada de los tipos de cada elemento sin ser por la estructura que se impone en su tipo. El tipo utilizado en este trabajo (al igual que el tipo `HList` utilizado por Idris) permite utilizar tipos arbitrarios y obtener información de ellos accediendo a la lista de tipos, por lo cual son más útiles.

Algunos ejemplos son los siguientes:

```
hListTuple : HList (\x => (x, x))
hListTuple = (1,1) :: ("1","2") :: Nil

hListExample : HList id
hListExample = 1 :: "1" :: (1,2) :: Nil
```

Como se ve en el último ejemplo, se puede reconstruir la definición de HList existencial simple utilizando `HList id`.

Capítulo 4

Caso de estudio

PENDIENTE

Capítulo 5

Trabajo a futuro

PENDIENTE

Capítulo 6

Conclusión

PENDIENTE

Capítulo 7

Apéndice

7.1. Código fuente

PENDIENTE

Bibliografía

[1] *Elm - Compilers as Assistants*. 2015. URL: http://elm-lang.org/blog/compilers-as-assistants (visitado 17-04-2016).

[2] *Elm - Records*. 2016. URL: http://elm-lang.org/docs/records (visitado 08-03-2016).

[3] Benedict R. Gaster y Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. Inf. téc. 1996.

[4] Wolfgang Jeltsch. «Generic Record Combinators with Static Type Checking». En: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP '10. Hagenberg, Austria: ACM, 2010, págs. 143-154. ISBN: 978-1-4503-0132-9. DOI: 10.1145/1836089.1836108. URL: http://doi.acm.org/10.1145/1836089.1836108.

[5] Oleg Kiselyov, Ralf Lämmel y Keean Schupke. *Hackage - The HList package*. 2004. URL: https://hackage.haskell.org/package/HList (visitado 06-03-2016).

[6] Oleg Kiselyov, Ralf Lämmel y Keean Schupke. «Strongly Typed Heterogeneous Collections». En: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, 2004, págs. 96-107. ISBN: 1-58113-850-4. DOI: 10.1145/1017472.1017488. URL: http://doi.acm.org/10.1145/1017472.1017488.

[7] Daan Leijen. «Extensible records with scoped labels». En: *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*. Tallin, Estonia, sep. de 2005.

[8] Daan Leijen. *First-class labels for extensible rows*. Inf. téc. UU-CS-2004-51. Department of Computer Science, Universiteit Utrecht, dic. de 2004.

[9] Bruno Martinez, Marcos Viera y Alberto Pardo. «Just Do It While Compiling!: Fast Extensible Records in Haskell». En: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*. PEPM '13. Rome, Italy: ACM, 2013, págs. 77-86. ISBN: 978-1-4503-1842-6. DOI: 10.1145/2426890.2426908. URL: http://doi.acm.org/10.1145/2426890.2426908.

[10] *Purescript - Handling Native Effects with the Eff Monad*. 2016. URL: http://www.purescript.org/learn/eff/ (visitado 17-04-2016).