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

Pendiente: Ingresar introduccion

Capítulo 2

Listas Heterogéneas

2.1. Introducción

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 heterogéneas' 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. 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. HList en Haskell

Esta forma de crear listas heterogéneas es presentada en [3], e implementada en el paquete *hlist*, encontrado en Hackage [2].

En esta libería, HList se define 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 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`. Sin embargo, la capacidad de Haskell para usar tipos dependientes no es del todo avanzada, por lo cual no es posible, por ejemplo, definir `HList` como un tipo común, y no como un *data family*. Esto fuerza a uno a crear funciones sobre este tipo de listas utilizando otros medios, como *type families* como se muestra aquí:

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

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, lo cual no sucede en el caso de HList de Haskell.

2.3. HList en Idris

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` (que representa un vector de naturales de 2 elementos).

La definición de HList utilizada en Idris es la siguiente:

```
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)
```

Esta definición es utilizada para poder definir los records extensibles más adelante, por lo que no solo permite tener una lista de elementos de cualquier tipo, sino que también permite asociarle una etiqueta a cada uno de esos tipos.

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 tipo `HList` se define como una función de tipo que toma un LabelList y retorna un tipo. Es decir, es un tipo parametrizado por un tipo `lty` (este tipo está implícito en la definición anterior y puede ser cualquiera), y un valor del tipo `LabelList lty`. La definición del tipo se realiza enumerando sus constructores, el cual, al ser una lista, se corresponde a los constructores generales de éstas: un constructor para la lista vacía y uno para agregar un elemento a su cabeza. El constructor `Nil` simplemente identifica una lista heterogénea vacía. Al estar vacía no existen tipos que pertenezcan a ella, por lo cual la LabelList en su tipo es vacía también. El constructor `(::)` es un constructor infijo que representa a `Cons`. Este constructor toma una etiqueta, un valor y una HList, y agrega ese valor a la lista. A su vez, agrega el tipo de ese valor, junto a su etiqueta asociada, a la LabelList del tipo de HList.

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

```
23 :: "Hello World" :: [1,2,3] :
    HList [("Etiqueta1",Nat), ("Etiqueta2", String), ("Etiqueta3", [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.

2.4. Otras alternativas

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 ellas.

2.4.1. HVect

```
data HVect : Vect k Type -> Type where
  Nil : HVect []
  (::) : t -> HVect ts -> HVect (t::ts)
```

HVect es un tipo que viene incluido en el Prelude de Idris, y es la implementación más sencilla y estándar de listas heterogéneas en Idris. El enfoque que se decidió utilizar se basa mucho en esta definición, con algunas diferencias:

- Se utiliza el tipo `List` en vez de `Vect`, ya que no es necesario mantener la información del largo de la lista/vector.
- En vez de que la lista de tipos solo tenga el tipo, se decidió que también contenga una etiqueta asociada a tal, para poder ser utilizado para definir records extensibles.

Un ejemplo de su uso es

```
["Hello",[1,2,3],42,(0,10)] :
    HVect [String, List Nat, Nat, (Nat, Nat)]
```

2.4.2. 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).

Un ejemplo de su uso es

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

2.4.3. 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.

Un ejemplo de su uso es

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

2.4.4. 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 HVect) 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. Otras listas heterogéneas como `HVect` no imponen ninguna restricción, por lo que son más poderosas.

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 simple utilizando `HList id`

Capítulo 3

Records Extensibles

3.1. Introducción

Para analizar la capacidad para definir DSLs de un lenguaje con tipos dependientes como Idris se decidió investigar la implementación de records extensibles en Idris.

La necesidad de tener records extensibles ocurre de un problema que tienen otros lenguajes estáticos en relación a records: ¿Cómo puedo modificar la estructura de un record ya definido?

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 record de manera estática, indicando su tipo, su nombre, etc.

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. 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.

Poder implementar records extensibles en un lenguaje no es trivial, y en algunos casos implica incluir esta funcionalidad a nivel de lenguaje, agregando construcciones

específicas al lenguaje que permitan definir este tipo de records, como es en el caso de Elm [1] y otros lenguajes. En este trabajo se va a mostrar cómo poder definir records extensibles utilizando las construcciones y funcionalidades que Idris ya tiene, y poder definirlas como una librería, haciendo uso de tipos dependientes.

3.2. Records Extensibles en Idris

Un record necesita ser una estructura de datos que permita tener una cantidad arbitraria de valores. Como se indicó anteriormente, también es necesario que estos valores tengan una etiqueta o nombre asociado para poder referenciarlos fuera del record. También es necesario conocer el tipo de tal valor, tal que cuando se referencie tal valor (utilizando su etiqueta), es posible conocer su tipo para saber cómo operar sobre él. El tipo `HList` definido en el capítulo anterior cumple con todas estas características. Al ser una lista heterogénea, HList permite agrupar una cantidad arbitraria de valores con tipos distintos, el cual es el requerimiento principal de un record. A su vez, cada tipo tiene asociado una etiqueta, por lo que uno puede referenciar la posición en la lista que ocupa un valor (y por lo tanto obtener el valor mismo), y su tipo, solo referenciando su etiqueta. Por lo tanto una lista heterogénea, con la definición de `HList`, es un candidato para poder representar records extensibles en Idris.

Sin embargo surgen dos problemas. Primero, las etiquetas no pueden ser cualquier tipo arbitrario (como lo son en `HList`), sino que éstas deben tener un tipo que permita compararlas. Para poder referenciar las etiquetas fuera del record, es necesario poder comparar con igualdad cada etiqueta del record con la que se intenta referenciar, para poder saber cuál valor se quiere obtener o manipular. Segundo, para poder referenciar un valor dada su etiqueta, es necesario que tal etiqueta sea única en el record. De forma contraria, cada vez que se intenta acceder a un valor se introduce ambigüedad, al no saber cuál de las etiquetas idénticas utilizar.

El primer problema tiene dos potenciales soluciones. Una de ellas es forzar a que el tipo de etiquetas sea `String`. `String` es un tipo que es posible comparar con igualdad, y a su vez es el tipo estándar para definir etiquetas y nombres. Sin embargo, aquí se decidió utilizar un enfoque más general. Se decidió utilizar un tipo genérico `lty` como etiqueta, pero con la restricción de que pertenezca a la typeclass `DecEq lty`. Esta typeclass se caracteriza por definir la siguiente operación

```
decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
```

Esta operación permite verificar la igualdad de dos valores de tal tipo. El tipo resultante `Dec (x1 = x2)` indica que es posible encontrar una prueba de que x1 es idéntico a x2, o es posible encontrar una prueba de que no son idénticos (llamado tipo de *decisión*). Al aplicar la restricción `DecEq lty` es posible realizar chequeos de igualdad (a nivel de tipos) de las etiquetas.

Para resolver el segundo problema es necesario poder afirmar que la lista de etiquetas de un record es efectivamente un conjunto, es decir que no tiene repetidos. En Idris es posible codificar esta propiedad en un tipo de datos:

```
data IsSet : List t -> Type where
  IsSetNil : IsSet []
  IsSetCons : Not (Elem x xs) -> IsSet xs -> IsSet (x :: xs)
```

La correspondencia de Curry-Howard nos permite asegurar que un tipo se corresponde a un teorema o proposición lógica, y valores que instancian ese tipo se corresponden a pruebas de ese teorema. Para el caso de conjuntos, se puede ver a la propiedad de no tener elementos repetidos como un predicado sobre una lista de tales elementos, donde `IsSet ls` indica que la lista `ls` no tiene elementos repetidos.

Las pruebas de este predicado se construyen de forma constructiva. Primero se prueba que la lista vacía no contiene repetidos, y luego para el caso recursivo si se agrega un elemento a una lista, la lista resultante no va a tener repetidos solamente si el elemento a agregar no se encuentra en la lista original.

Como para el caso de records extensibles se manejan listas de pares de etiquetas y tipos, se definen los siguientes tipos y funciones útiles para manejarlos:

```
labelsOf : LabelList lty -> List lty
labelsOf = map fst

IsLabelSet : LabelList lty -> Type
IsLabelSet ts = IsSet (labelsOf ts)
```

El predicado `IsLabelSet ls` indica que para la lista de etiquetas y tipos `ls` no existen etiquetas repetidas.

Obtener una definicion de record extensible es simplemente necesitar una lista heterogenea etiquetada, y una prueba de que las etiquetas no son repetidas.

```
data Record : LabelList lty -> Type where
    MkRecord : IsLabelSet ts -> HList ts -> Record ts
```

3.3. Funcionalidades de records

Dada esta definición de records es posible definir varias funcionalidades de éstos.
Como caso inicial es posible definir un record vacío

```
emptyRec : Record []
emptyRec = MkRecord IsSetNil {ts=[]} []
```

A su vez se pueden crear funciones que proyectan sobre los campos del constructor del record. En particular, se puede convertir un record a un HList, y dado un record se puede obtener la prueba de que sus etiquetas forman un conjunto.

```
recToHList : Record ts -> HList ts
recToHList (MkRecord _ hs) = hs

recLblIsSet : Record ts -> IsLabelSet ts
recLblIsSet (MkRecord lsIsSet _ ) = lsIsSet
```

Sin embargo, la funcionalidad más importante es la de poder extender tal record con un nuevo campo. Esta funcionalidad puede definirse de esta forma

```
consRec : DecEq lty => {ts : LabelList lty} -> {t : Type} ->
    (lbl : lty) -> (val : t) ->  Record ts ->
    {notElem : Not (ElemLabel lbl ts)} -> Record ((lbl,t) :: ts)
consRec lbl val (MkRecord subLabelSet hs) {notElem} =
    MkRecord (IsSetCons notElem subLabelSet) (val :: hs)
```

Antes que nada se necesita que el tipo de etiquetas admita igualdad, utilizando la restricción `DecEq lty`. Luego, se necesita un record ya existente `Record ts`, el nuevo valor `val` y la etiqueta del nuevo campo `lbl`. A su vez es necesaria una prueba de que la etiqueta nueva no esté repetida en las etiquetas ya existentes del record, representado por el tipo `Not (ElemLabel lbl ts)`. Si se cumplen estas condiciones entonces es posible crear un nuevo record con el nuevo campo, el cual es reflejado en el tipo `Record ((lbl,t) :: ts)`, donde se agrega el par con la nueva etiqueta y el tipo del valor a la lista de tipos y etiquetas original.

PENDIENTE: hLookupByLabel

PENDIENTE: hProjectByLabels

PENDIENTE: Truco de TypeOrUnit

PENDIENTE: Diferencia de hProjectByLabels con 'projectLeft' y 'IsProjectLeft'

Bibliografía

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

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

[3] 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.