UNIVERSIDAD DE LA REPÚBLICA, URUGUAY

PROYECTO DE GRADO

Desarrollo de DSLs en lenguajes con tipos dependientes

Gonzalo Waszczuk

Supervisado por
Marcos VIERA
Alberto PARDO

Índice general

Introducción

Pendiente: Ingresar introduccion

Listas Heterogéneas

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.

HList en Haskell

Esta forma de crear listas heterogéneas es presentada en el paper Kiselyov et al. [2004b], e implementada en el paquete *hlist*, encontrado en Hackage [1].

En esta libería, HList se define de la siguiente forma:

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

[1] O. Kiselyov, R. Lämmel, and K. Schupke. Hackage - the hlist package, 2004a. URL https://hackage.haskell.org/package/HList

```
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.

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]]
```

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.

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.

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)]
```

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
```

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
```

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`

Records Extensibles

Introducción

PENDIENTE

Bibliografía

O. Kiselyov, R. Lämmel, and K. Schupke. Hackage - the hlist package, 2004a. URL https://hackage.haskell.org/package/HList.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004b. ACM. ISBN 1-58113-850-4. DOI: 10.1145/1017472.1017488. URL http://doi.acm.org/10.1145/1017472.1017488.