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 ante un problema que tienen lenguajes de programacion estáticos en cuanto a records: ¿Cómo puedo modificar la estructura de un record ya definido?

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. Como ejemplo se tiene la siguiente funcion

```
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 o igual 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, como podrian

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.

La version de Idris utilizada en este trabajo es la 0.12.0

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:

```
interface 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 interprete 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, donde a la lista se puede agregar un entero, un float y un texto.

```
'(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)
```

Type families como la anterior permiten que se tenga un tipo `HLength ls` en la definicion de una funcion y el typechecker decida cual de las instancias anteriores debe llamar, culminando en un valor del kind `HNat`.

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 evita que uno

tenga que recurrir al tipo de construcciones de Haskell (type families, etc) 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]]
```

Cada valor agregado a la lista tiene un tipo asociado que es almacenado en la lista del tipo. `23 : Nat` guarda `23` en la lista pero `Nat` en el tipo, de forma que uno siempre puede recuperar ya sea el tipo o el valor si se quieren utilizar 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. HList actualizado

En este trabajo se decidio implementar las listas heterogeneas de una forma distinta a la que utiliza Idris en general. Las listas heterogeneas 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 implementacion es la de `Record` de Haskell definida en el capítulo anterior. En esta 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 decidio no utilizar esta implementacion, ya que necesita definir tipos auxiliares (como `Tagged`), y a su vez necesita definir predicados específicos para esta implementacion, como el predicado que indica que la lista heterogenea está compuesta solamente por valores del tipo `Tagged`.

Se decidio utilizar una solucion 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 []
  (::) : {l : lty} -> (val : t) -> HList ts ->
    HList ((l,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 implementacion es identica a la de `HList` de Idris, con la diferencia de que se debe pasar no solo el valor sino la etiqueta de su campo tambien.

En Idris se pueden definir parametros implicitos, como `{l : lty}`, a diferencia de parametros explicitos como `(val : t)`. Al momento de utilizar un tipo o una funcion no es necesario incluir los parametros implicitos, Idris va a intentar calcularlos

automaticamente basados en el contexto de la llamada o uso del tipo o funcion. Si Idris no puede calcularlos, el typechecker va a fallar y va a ser necesario explicitar tales parametros de esta forma: `{parametro=valor}`.

3.2. Definicion de Records Extensibles

Al igual que la implementacion de records extensibles en Haskell vista en la seccion anterior, en este trabajo se decidio implementar records extensibles utilizando una lista heterogenea, cuyo tipo contiene una lista de etiquetas que no deben ser repetidas.

Para comenzar con esta implementacion, se necesita que las etiquetas de los records puedan ser comparadas para verificar que no sean repetidas. En este trabajo se decidio utilizar la *typeclass* `DecEq lty` para las etiquetas de tipo `lty`. Al ser las etiquetas de un tipo que tienen una instancia de esa typeclass, la igualdad de cualquier par de valores de etiquetas es decidible, por lo que las etiquetas pueden ser comparadas.

Como en la implementacion en Haskell se decide utilizar un predicado que indica que una lista de valores 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)
```

El tipo `IsSet ls` funciona efectivamente como un predicado logico, el cual indica que la lista `ls` es un conjunto que 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:

```
ElemLabel : lty -> LabelList lty -> Type
ElemLabel l ts = Elem l (labelsOf ts)

isElemLabel : DecEq lty => (l : lty) ->
  (ts : LabelList lty) ->
  Dec (Elem l (labelsOf ts))
isElemLabel l ts = isElem l (labelsOf ts)

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

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

El predicado `IsLabelSet ts` indica que para la lista de etiquetas y tipos `ts` no existen etiquetas repetidas. El predicado `ElemLabel l ts` indica que la etiqueta `l` pertenece a la lista de etiquetas `ts`. `isElemLabel` es una funcion de decision, donde para cualquier etiqueta y cualquier lista, se puede probar que tal etiqueta pertenece a esa lista o no.

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
```

Un caso base que se puede definir es el 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
```

3.3. Extension de records dinámicamente

La funcionalidad más importante de records extensibles 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} -> (l : lty) -> (val : t) ->
  Record ts -> {notElem : Not (ElemLabel l ts)} ->
  Record ((l,t) :: ts)
consRec l 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 restriccion `DecEq lty`. Luego, se necesita un record ya existente `Record ts`, el nuevo valor `val` y la etiqueta del nuevo campo `l`. A su vez es necesaria una prueba de que la etiqueta nueva no este repetida en las etiquetas ya existentes del record, representado por el tipo `Not (ElemLabel l 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 ((l,t) :: ts)`, donde se agrega el par con la nueva etiqueta y el tipo del valor a la lista de tipos y etiquetas original.

3.3.1. Construccion de terminos de prueba

El principal problema que ocurre al utilizar esa funcion se da por el parámetro `notElem`. Para poder llamar a esta funcion es necesario construir una prueba de que la nueva etiqueta a agregar al record no está repetida en el record ya existente.

Una primera opcion es generar la prueba manualmente, pero esto resulta tedioso e impráctico, ya que sería necesario construir ese termino de prueba cada vez que se llame a la funcion.

Una segunda opcion es utilizar la decidibilidad del predicado a instanciar. Al tener un tipo decidible, es posible utilizar un truco del typechecker forzando la unificacion del tipo decidible con el tipo mismo o su contradiccion. Básicamente, si uno puede generar un valor del tipo `Dec p`, entonces puede unificarlo con `p` en tiempo de compilacion, o con `Not p`. Si en tiempo de compilacion la unificacion falla, entonces el typechecker falla.

En este caso en particular, es necesario poder obtener un valor de tipo `Dec (ElemLabel l ts)`, lo cual puede hacerse con la funcion `isElemLabel`.

Para forzar la unificacion, se utilizan estas funciones auxiliares:

```
getYes : (d : Dec p) ->
  case d of { No _ => (); Yes _ => p}
getYes (No _ ) = ()
getYes (Yes yes) = yes

getNo : (d : Dec p) ->
  case d of { No _ => Not p; Yes _ => ()}
getNo (No no) = no
getNo (Yes _ ) = ()
```

Se analizará el caso de `getYes` primero, y luego se aplicará el mismo razonamiento para `getNo`. `getYes` toma un tipo `Dec p` y el tipo que retorna es una computacion la cual hace *pattern matching* sobre el valor de `Dec p`. Esta computacion es ejecutada en tiempo de typechecking, y tiene dos opciones: o retorna el tipo *top* `()`, o retorna el tipo `p`. En tiempo de typechecking, se hace pattern matching sobre `Dec p`. Si la prueba es de `No`, entonces se retorna el valor `()` (que efectivamente es un valor del tipo `()`). Sin embargo, si la prueba es de `Yes`, entonces ya se tiene un valor de tipo `p`, por lo cual se retorna ese. El pattern matching no solo permite tener una bifurcacion sobre cuál valor retornar, sino tambien sobre cuál es el tipo de este valor retornado, pudiendo retornar dos valores con tipos totalmente distintos.

La funcion `getNo` es identica, pero retornando un valor del tipo `Not p`.

A continuacion se muestran ejemplos del uso de estas funciones:

```
okYes : Elem "L1" ["L1"]
okYes = getYes $ isElem "L1" ["L1"]

okNo : Not (Elem "L1" ["L2"])
okNo = getNo $ isElem "L1" ["L2"]

badYes : Elem "L1" ["L2"]
badYes = getYes $ isElem "L1" ["L2"]
```

```
badNo : Not (Elem "L1" ["L1"])
badNo = getNo $ isElem "L1" ["L1"]
```

En el caso de `okYes`, la funcion `isElem` es computada en tiempo de typechecking, retornando la prueba de `Elem "L1"["L1"]`. Luego `getYes` hace pattern matching sobre tal valor y encuentra que se corresponde al caso de `Yes`, por lo cual retorna ese mismo valor del tipo `Elem "L1"["L1"]`. No ocurre lo mismo para el caso de `badYes`, donde en tiempo de typechecking se retorna la prueba de `Not (Elem "L1"["L2"])`. Al realizar pattern matching entonces `getYes` retorna `()`, lo cual no puede ser unificado con `Elem "L1"["L2"]`, mostrando error de typechecking.

Lo mismo ocurre de forma inversa con `okNo` y `badNo`.

Con este truco se puede generar una prueba automática de cualquier predicado decidible, por lo que se puede simplificar el uso de `consRec`. Ahora puede ser utilizado de esta forma:

```
extendedRec : Record [("Nombre", String)]
extendedRec = consRec "Nombre" "Juan"
  {notElem=(getNo $ isElemLabel "Nombre" [])} emptyRec
```

3.3.2. Generacion de pruebas automática

Al utilizar `getYes` y `getNo` se simplifica bastante el proceso de construccion de pruebas, pero de todas formas se necesita llamar a esas funciones manualmente. Es posible mejorar este sistema.

A continuacion se muestra un nuevo truco que utiliza el mismo concepto del anterior, donde se realiza pattern matching sobre un tipo decidible en tiempo de typechecking para unificar tipos. Sin embargo, el pattern matching se realiza en el tipo mismo y no en una funcion auxiliar.

Esto es posible gracias a este tipo y esta funcion:

```
TypeOrUnit : Dec p -> Type -> Type
TypeOrUnit (Yes yes) res = res
TypeOrUnit (No _) _ = ()

mkTypeOrUnit : (d : Dec p) -> (cnst : p -> res) ->
  TypeOrUnit d res
mkTypeOrUnit (Yes prf) cnst = cnst prf
mkTypeOrUnit (No _) _ = ()
```

El tipo `TypeOrUnit` permite discriminar un tipo en dos casos:

- Si `Dec p` incluye una prueba de `p`, entonces se obtiene el tipo deseado.
- Si `Dec p` incluye una contradiccion de `p`, entonces se obtiene el tipo `()`

Este metodo funciona cuando se necesita unificar un tipo `type` y `TypeOrUnit dec type`. Si se tiene una prueba de `p` entonces la unificacion va a dar correcta, pero si no se tiene una prueba entonces va a fallar el typechecking.

`mkTypeOrUnit` es el constructor de este tipo. Necesita la prueba o contradiccion de `p` y una funcion que construya el tipo deseado dada una prueba de `p`. Este constructor es utilizado solamente cuando se tiene tal prueba.

Como ejemplo, se tiene una funcion `addNat` que debe agregar un natural a una lista solamente si ya pertenece a esta, y de caso contrario tirar error de typechecking.

```
-- isElem : DecEq a => (x : a) -> (xs : List a) ->
--  Dec (Elem x xs)
addNat : (n : Nat) -> (ns : List Nat) ->
  TypeOrUnit (isElem n ns) (List Nat)
addNat n ns = mkTypeOrUnit (isElem n ns)
  (\isElem => Prelude.List.(::) n ns)

myListOk : List Nat
myListOk = addNat 10 [10] -- [10, 10]

myListBad1 : List Nat
myListBad1 = addNat 9 [10] -- Error de typechecking

myListBad2 : Nat -> List Nat
myListBad2 n = addNat n [10] -- Error de typechecking
```

La funcion `addNat` hace uso de `TypeOrUnit`, forzando que se cumpla el predicado `Elem n ns`. Si no se cumple la funcion retorna `()`. La unificacion se realiza en la llamada en `myListOk`. La llamada a `addNat [10] 10` retorna el tipo `TypeOrUnit (isElem 10 [10]) (List Nat)`, pero `myListOk` espera `List Nat`. En tiempo de typechecking se evalúa `isElem 10 [10]`, el cual retorna una prueba de `Elem 10 [10]`, por lo que `TypeOrUnit (isElem 10 [10]) (List Nat)` evalúa a `List Nat`, compilando correctamente el codigo.

En el caso de `myListBad1`, como `isElem 9 [10]` retorna una contradiccion de `Elem 9 [10]`, `TypeOrUnit (isElem 9 [10]) (List Nat)` evalúa a `()`, el cual no puede ser unificado con `List Nat`, tirando un error de typechecking.

Otro caso de error ocurre con `myListBad2`. En este caso es imposible evaluar completamente `isElem n [10]`, ya que no se conoce el valor de `n` en tiempo de typechecking. Por lo tanto no se puede evaluar `TypeOrUnit (isElem n [10]) (List Nat)`, lo cual hace que falle la unificacion con `List Nat`.

Por lo tanto, con este metodo es posible forzar, en tiempo de compilacion, a que se cumpla un predicado específico. Si el predicado puede ser evaluado y es correcto, entonces el codigo compila correctamente. Si el predicado no puede ser evaluado, o puede pero resulta ser incorrecto, entonces el codigo no compila.

Para poder aplicar este metodo a los records, es necesario poder tener una funcion que obtenga una prueba o contradiccion de que los campos del record no tienen repetidos. Esa funcion es la siguiente:

```
ifNotSetHereThenNeitherThere : Not (IsSet xs) ->
  Not (IsSet (x :: xs))
ifNotSetHereThenNeitherThere notXsIsSet
  (IsSetCons xIsInXs xsIsSet) = notXsIsSet xsIsSet
```

```
ifIsElemThenConsIsNotSet : Elem x xs ->
  Not (IsSet (x :: xs))
ifIsElemThenConsIsNotSet xIsInXs
  (IsSetCons notXIsInXs xsIsSet) = notXIsInXs xIsInXs

isSet : DecEq t => (xs : List t) -> Dec (IsSet xs)
isSet [] = Yes IsSetNil
isSet (x :: xs) with (isSet xs)
  isSet (x :: xs) | No notXsIsSet =
    No $ ifNotSetHereThenNeitherThere notXsIsSet
  isSet (x :: xs) | Yes xsIsSet with (isElem x xs)
    isSet (x :: xs) | Yes xsIsSet | No notXInXs =
      Yes $ IsSetCons notXInXs xsIsSet
    isSet (x :: xs) | Yes xsIsSet | Yes xInXs =
      No $ ifIsElemThenConsIsNotSet xInXs

isLabelSet : DecEq lty => (ts : LabelList lty) ->
  Dec (IsLabelSet ts)
isLabelSet ts = isSet (labelsOf ts)
```

`ifNotSetHereThenNeitherThere` y `ifIsElemThenConsIsNotSet` son dos lemas necesarios para poder definir `isSet`. `isSet` toma una lista de valores que pueden chequearse por igualdad, y retorna o una prueba de que no tiene repetidos o una prueba de que los hay.

`isLabelSet` simplemente permite aplicar la funcion `isSet` al tipo `IsLabelSet`.

La implementacion de `isSet` realiza un análisis de casos sobre el largo de la lista. Para el caso de lista vacía esta no tiene elementos repetidos por definicion. Para el caso de que tenga un elemento seguido de la cola de la lista, realiza dos análisis de casos seguidos, verificando si la cola de la lista no tiene repetidos (utilizando recursion), y luego verificando que la cabeza de la lista no pertenezca a la cola de esta. En algunos casos utiliza los lemas definidos previamente si es necesario.

En Idris un análisis de casos que tiene impacto en los tipos utiliza el identificador `with`. Su sintaxis es del estilo

```
func params with (expresion)
  func params | Caso1 val1 = ...
  func params | Caso2 val2 = ...
```

La expression dentro del `with` es deconstruida en sus constructores correspondientes. La diferencia con `case` es que `with` permite redefinir los parámetros anteriores según el resultado del matcheo de la expresion. Al ser Idris un lenguaje con tipos dependientes pueden ocurrir situaciones donde una expresion matchee solamente cuando otros valores previos de la definicion tienen valores fijos. Un ejemplo es el siguiente:

```
eq : (n : Nat) -> (m : Nat) -> Bool
eq n m  with (decEq n m)
  eq n n | Yes Refl = True
  eq n m | No notNEqM = False
```

El matcheo de `Yes` tiene un valor de tipo `val : n = m`. Como la única forma de que se tenga una prueba de ambos es que `n` sea efectivamente `m`, se puede unificar `val` con `Refl : n = n` y cambiar `eq n m` por `eq n n` en la definicion de la funcion.

Luego de tener la funcion de decidibilidad anterior definida, es posible aplicar el tipo `TypeOrUnit` a los records de la siguiente forma:

```
RecordOrUnit : DecEq lty => LabelList lty -> Type
RecordOrUnit ts = TypeOrUnit (isLabelSet ts) (Record ts)
```

`RecordOrUnit ts` evalúa a `Record ts` cuando se cumple que `ts` no tiene repetidos, pero evalúa a `()` cuando tiene repetidos.

Con este tipo es posible tener la siguiente funcion que extiende un record:

```
consRecAuto : DecEq lty => {ts : LabelList lty} ->
  {t : Type} -> (l : lty) -> (val : t) -> Record ts ->
  RecordOrUnit ((l,t) :: ts)
consRecAuto {ts} {t} l val (MkRecord tsIsLabelSet hs) =
  mkTypeOrUnit (isLabelSet ((l, t) :: ts))
  (\isLabelSet => MkRecord isLabelSet (val :: hs))
```

Esta funcion es identica a `consRec`, solamente que no es necesario pasar una prueba de `Not (ElemLabel l ts)`. Ahora se calcula automáticamente la prueba de `IsLabelSet ((l,t) :: ts))` en tiempo de typechecking y se impacta en el tipo resultante `RecordOrUnit ((l,t) :: ts)`.

Con esta nueva funcion es posible extender un record de la siguiente manera:

```
extendedRec1 : Record [("Nombre", String)]
extendedRec1 = consRecAuto "Nombre" "Juan" emptyRec

extendedRec2 : Record [("Edad", Nat), ("Nombre", String)]
extendedRec2 = consRecAuto "Edad" 24 extendedRec1
```

Otra funcion útil que se puede definir con este metodo es la que toma una lista heterogenea y la convierte en un record:

```
hListToRecAuto : DecEq lty => (ts : LabelList lty) ->
  HList ts -> RecordOrUnit ts
hListToRecAuto ts hs = mkTypeOrUnit (isLabelSet ts)
  (\tsIsSet => MkRecord tsIsSet hs)
```

Esta funcion permite construir records solamente indicando su lista de campos, y luego su lista de valores. Un ejemplo es el siguiente:

```
rec : Record [("Apellido", String), ("Nombre", String)]
rec = hListToRecAuto
  [("Apellido", String), ("Nombre", String)]
  ["Sanchez", "Juan"]
```

3.4. Proyeccion de un record

Dado un record, una funcionalidad importante que se puede tener es poder filtrar el record dada una lista de etiquetas, donde es posible obtener un nuevo record con solo los campos indicados en una lista de etiquetas proporcionada por el usuario. Esto se llama proyeccion de un record, y es otra funcionalidad clave de los records extensibles.

Un ejemplo de su uso seria el siguiente

```
rec : Record [("Nombre", String), ("Edad", Nat),
              ("Apellido", String)]
rec = consRecAuto "Nombre" "Juan" $
              consRecAuto "Edad" 24 $
              consRecAuto "Apellido" "Sanchez" $
              emptyRec

projectedRec: Record [("Apellido", String), ("Edad", Nat)]
projectedRec = hProjectByLabelsAuto ["Apellido", "Edad"] rec
-- ["Sanchez", 24]
```

`hProjectByLabelsAuto` en este caso toma una lista de etiquetas y retorna un subconjunto del record con solo esas etiquetas.

El uso de `$` permite aplicar una funcion a sus argumentos sin tener que usar parentesis. Esta sintaxis proviene de Haskell y es definida de esta manera:

```
($) : (a -> b) -> a -> b
($) f a = f a
```

Esta herramienta facilita la aplicacion de funciones gracias a su precedencia, donde `f x (g y (h z))` es equivalente a `f x $ g y $ h z`.

Para comenzar a definir proyeccion sobre records se necesita definir un predicado que indique que una lista es el resultado de eliminar un elemento de otra.

```
data DeleteElemPred : (xs : List t) -> Elem x xs ->
  List t -> Type where
  DeleteElemPredHere : DeleteElemPred (x :: xs) Here xs
  DeleteElemPredThere : {isThere : Elem y xs} ->
    DeleteElemPred xs isThere ys ->
    DeleteElemPred (x :: xs) (There isThere) (x :: ys)
```

El tipo `DeleteElemPred xs elem ys` toma dos listas `xs` y `ys`, una prueba `elem` de que un elemento pertenece a `xs`, y representa el predicado de que la lista `ys` es el resultado de eliminar ese elemento de `xs`. El caso base del predicado es `DeleteElemPredHere`, donde el elemento a eliminar es el primero de la lista. El caso recursivo es `DeleteElemPredThere`, donde elimina el elemento del resto de la lista y deja el primer elemento sin cambiar.

La función que computa la lista resultado que es equivalente al predicado anterior es

```
deleteElem : (xs : List t) -> Elem x xs -> List t
deleteElem (x :: xs) Here = xs
deleteElem (x :: xs) (There inThere) =
  let rest = deleteElem xs inThere
  in x :: rest
```

En un lenguaje como Idris se pueden definir funciones que computen un resultado, o un predicado que establezcan que uno de sus parametros sea ese mismo resultado.

Estas dos formas de definir una funcionalidad son equivalentes, lo cual se prueba definiendo las siguientes dos funciones:

```
fromDeleteElemPredToComp : {xs1, xs2 : List t} ->
  {isElem : Elem x xs1} -> DeleteElemPred xs1 isElem xs2 ->
  xs2 = deleteElem xs1 isElem

fromCompToDeleteElemPred : (xs : List t) ->
  (isElem : Elem x xs) ->
  DeleteElemPred xs isElem (deleteElem xs isElem)
```

Estas funciones permiten establecer un isomorfismo entre `deleteElem xs isElem` y `DeleteElemPred xs isElem ys`.

Para completar la definicion del tipo se define la funcion de decibilidad de esta, con el siguiente tipo

```
isDeleteElemPred : DecEq t => (xs : List t) ->
  (isElem : Elem x xs) -> (res : List t) ->
  Dec (DeleteElemPred xs isElem res)
```

Con esta definicion se pueden definir los predicados de proyeccion

```
data IsProjectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty -> Type where
  IPL_EmptyLabels : DecEq lty => IsProjectLeft {lty} [] ts []
  IPL_EmptyVect : DecEq lty => IsProjectLeft {lty} ls [] []
  IPL_ProjLabelElem : DecEq lty => (isElem : Elem l ls) ->
    DeleteElemPred ls isElem lsNew ->
    IsProjectLeft {lty} lsNew ts res1 ->
    IsProjectLeft ls ((l,ty) :: ts) ((l,ty) :: res1)
  IPL_ProjLabelNotElem : DecEq lty => Not (Elem l ls) ->
    IsProjectLeft {lty} ls ts res1 ->
    IsProjectLeft ls ((l,ty) :: ts) res1

data IsProjectRight : DecEq lty => List lty -> LabelList lty ->
  LabelList lty -> Type where
  IPR_EmptyLabels : DecEq lty => IsProjectRight {lty} [] ts ts
  IPR_EmptyVect : DecEq lty => IsProjectRight {lty} ls [] []
  IPR_ProjLabelElem : DecEq lty => (isElem : Elem l ls) ->
    DeleteElemPred ls isElem lsNew ->
    IsProjectRight {lty} lsNew ts res1 ->
    IsProjectRight ls ((l,ty) :: ts) res1
```

```
IPR_ProjLabelNotElem : DecEq lty => Not (Elem l ls) ->
  IsProjectRight {lty} ls ts res1 ->
  IsProjectRight ls ((l,ty) :: ts) ((l,ty) :: res1)
```

`IsProjectLeft ls ts1 ts2` indica que la proyeccion de las etiquetas `ls` sobre la lista `ts1` equivale a la lista `ts2`. `IsProjectRight ls ts1 ts2` indica que al proyectar las etiquetas `ls` sobre la lista `ts1`, las etiquetas que no pertenecen a la proyeccion son las de la lista `ts2`. Se le llama proyeccion por izquierda y derecha ya que dada una lista de etiquetas esta siempre se puede dividir en dos al ser proyectada por otra, donde en la izquierda estan las etiquetas que pertenecen a la lista a proyectar y en la derecha las que no.

Algunos ejemplos son los siguientes

```
IsProjectLeft ["Edad"] [("Edad", Nat),("Nombre",String)]
  [("Edad",Nat)]
IsProjectRight ["Edad"] [("Edad", Nat),("Nombre",String)]
  [("Nombre", String)]
```

Al momento de definir estos tipos, el caso de `IsProjectLeft` tiene estos posibles casos

- Si se intenta proyectar por una lista vacia, la proyeccion de esta sobre cualquier otra lista va a ser vacia tambien, indicado por el tipo `IsProjectLeft lty [] ts []`
- Si se intenta proyectar hacia una lista vacia, entonces el resultado va a ser vacio tambien, indicado por el tipo `IsProjectLeft lty ls [] []`
- Si la lista donde se proyecta es del estilo `(l,ty) :: ts`, y si se tiene que esa etiqueta pertenece a la otra lista, con `isElem : Elem l ls`, entonces para proyectar la otra lista se debe eliminar ese elemento de esa con `DeleteElemPred ls isElem lsNew` obteniendo una nueva lista `lsNew` sin ese elemento. Luego se debe proyectar el resto de la lista con `IsProjectLeft lty lsNew ts res1`. La proyeccion final equivale a ese resultado mas el elemento eliminado, con el tipo `IsProjectLeft ls ((l,ty) :: ts) ((l,ty) :: res1)`
- Si la lista donde se proyecta es del estilo `(l,ty) :: ts`, y si se tiene que esa etiqueta no pertenece a la otra lista, con `Not (Elem l ls)`, entonces la proyeccion equivale a proyectar sobre el resto de la lista de la forma `IsProjectLeft lty ls ts res1`, retornando `IsProjectLeft ls ((l,ty) :: ts) res1`

Para el caso de `IsProjectRight` los casos son similares, donde la diferencia cae en que si se va a proyectar por un elemento que existe en la lista origen, entonces no se agrega a la lista destino (y si no pertenece a la lista origen si se agrega a la lista destino).

Al igual que `DeleteElemPred`, se puede definir una funcion que compute la proyeccion misma. Para el caso de `IsProjectLeft` (no vamos a enforcarnos en `IsProjectRight` de ahora en mas) esta es

```
projectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty
projectLeft [] ts = []
projectLeft ls [] = []
projectLeft ls ((l,ty) :: ts) with (isElem l ls)
  projectLeft ls ((l,ty) :: ts) | Yes lIsInLs =
    let delLFromLs = deleteElem ls lIsInLs
        rest = projectLeft delLFromLs ts
    in (l,ty) :: rest
  projectLeft ls ((l,ty) :: ts) | No _ = projectLeft ls ts
```

La implementacion de esta funcion sigue los mismos pasos de la definicion del predicado `IsProjectLeft`.

Tambien se puede definir un isomorfismo entre `projectLeft ls ts1` y `IsProjectLeft ls ts1 ts2` de esta forma

```
fromIsProjectLeftToComp : DecEq lty =>
  {ls : List lty} -> {ts1, ts2 : LabelList lty} ->
  IsProjectLeft ls ts1 ts2 ->
  IsSet ls -> ts2 = projectLeft ls ts1

fromCompToIsProjectLeft : DecEq lty => (ls : List lty) ->
  (ts : LabelList lty) ->
  IsProjectLeft ls ts (projectLeft ls ts)
```

En el caso de `fromIsProjectLeftToComp` es necesario tener una prueba de `IsSet ls`. Como las etiquetas de un record no pueden repetirse, no tiene sentido intentar proyectar por una lista con etiquetas repetidas. Sin embargo, en la definicion de varios tipos y funciones no es necesario explicitar esa precondicion de que no tengan etiquetas repetidas (proporcionando una prueba de `IsSet ls`). Solo es necesario proporcionar esa prueba en `fromIsProjectLeftToComp` porque necesita esa prueba en su implementacion. Su implementacion completa se encuentra en el apendice.

Dados estos tipos y funciones es posible definir una funcion que particione una lista heterogenea en dos, donde en el lado izquierdo se tienen los valores asociados a las etiquetas proyectadas y en el lado derecho los valores no asociados a tales

```
hProjectByLabelsHList : DecEq lty =>
  {ts : LabelList lty} -> (ls : List lty) ->
  HList ts ->
  ((ts1 : LabelList lty **
    (HList ls1, IsProjectLeft ls ts ts1)),
  (ts2 : LabelList lty **
    (HList ls2, IsProjectRight ls ts ts2)))
hProjectByLabelsHList [] {ts} hs =
  (([] ** ([], IPL_EmptyLabels)),
  (ts ** (hs, IPR_EmptyLabels)))
hProjectByLabelsHList _ [] =
  (([] ** ([], IPL_EmptyVect)),
  ([] ** ([], IPR_EmptyVect)))
```

```
hProjectByLabelsHList {lty} ls ((::) {l=l2} {t}
    {ts=ts2} val hs) =
  case (isElem l2 ls) of
    Yes l2InLs =>
      let
        (lsNew ** isDelElem) = deleteElemPred ls l2InLs
        ((subInLs ** (subInHs, subPrjLeft)),
          (subOutLs ** (subOutHs, subPrjRight))) =
          hProjectByLabelsHList {lty=lty} {ts=ts2} lsNew hs
        rPrjRight = IPR_ProjLabelElem {l=l2} {ty=t} {ts=ts2}
          {res1=subOutLs} l2InLs isDelElem subPrjRight
        rPrjLeft = IPL_ProjLabelElem {l=l2} {ty=t} {ts=ts2}
          {res1=subInLs} l2InLs isDelElem subPrjLeft
        rRight = (subOutLs ** (subOutHs, rPrjRight))
        rLeft = ((l2,t) :: subInLs ** (
          (::) {l=l2} val subInHs, rPrjLeft))
       in
         (rLeft, rRight)
    No notL2InLs =>
      let
        ((subInLs ** (subInHs, subPrjLeft)),
          (subOutLs ** (subOutHs, subPrjRight))) =
          hProjectByLabelsHList {lty=lty} {ts=ts2} ls hs
        rPrjLeft = IPL_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2}
          {res1=subInLs} notL2InLs subPrjLeft
        rPrjRight = IPR_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2}
          {res1=subOutLs} notL2InLs subPrjRight
        rLeft = (subInLs ** (subInHs, rPrjLeft))
        rRight = ((l2,t) :: subOutLs ** (
          (::) {l=l2} val subOutHs, rPrjRight))
      in
        (rLeft, rRight)
```

Se analizara esta funcion en partes:

```
hProjectByLabelsHList : DecEq lty => {ts : LabelList lty} ->
  (ls : List lty) -> HList ts ->
  ((ts1 : LabelList lty **
    (HList ts1, IsProjectLeft ls ts ts1)),
  (ts2 : LabelList lty **
    (HList ts2, IsProjectRight ls ts ts2)))
```

La definicion de la funcion toma una lista de etiquetas a proyectar `ls : List lty`, la lista heterogenea con valores `HList ts`, y retorna dos valores. En la izquierda retorna una lista `ts1 : LabelList lty` junto a la lista heterogenea `HList ts1`. A su vez retorna una prueba de `IsProjectLeft ls ts ts1`, que indica que al proyectar `ls` sobre `ts` el resultado fue `ts1`, es decir, `ts1` contiene las etiquetas proyectadas y `HList ts1` contiene los valores de esas etiquetas. El caso de la derecha es similar, pero teniendo una prueba de `IsProjectRight ls ts ts2`, lo cual indica que `HList ts2` tiene los valores de las etiquetas que no estan en `ls`.

```
hProjectByLabelsHList [] {ts} hs =
  (([] ** ([], IPL_EmptyLabels)),
  (ts ** (hs, IPR_EmptyLabels)))
```

Si se intenta proyectar por una lista vacia, entonces en la izquierda se retorna vacio pero en la derecha la HList original. Para las pruebas se utilizan las que corresponden a este caso base.

```
hProjectByLabelsHList _ [] =
  (([] ** ([], IPL_EmptyVect)),
  ([] ** ([], IPR_EmptyVect)))
```

Si la lista heterogenea origen es vacia, entonces ambos resultados van a ser vacios. Las pruebas utilizadas son las correspondientes a este caso base tambien.

```
hProjectByLabelsHList {lty} ls ((::) {l=l2} {t}
    {ts=ts2} val hs) =
  case (isElem l2 ls) of
    Yes l2InLs =>
```

Para el caso de que la HList sea del estilo `val :: hs` (se utiliza la notacion `(::) val hs` para poder utilizar las variables implicitas `l2` y `t`) se verifica si la etiqueta de ese primer valor pertenece a las que se quieren proyectar, con `isElem l2 ls : Elem l2 ls`.

```
Yes l2InLs =>
  let
    (lsNew ** isDelElem) = deleteElemPred ls l2InLs
    ((subInLs ** (subInHs, subPrjLeft)),
      (subOutLs ** (subOutHs, subPrjRight))) =
      hProjectByLabelsHList {lty=lty} {ts=ts2} lsNew hs
    rPrjRight = IPR_ProjLabelElem {l=l2} {ty=t} {ts=ts2}
      {res1=subOutLs} l2InLs isDelElem subPrjRight
    rPrjLeft = IPL_ProjLabelElem {l=l2} {ty=t} {ts=ts2}
      {res1=subInLs} l2InLs isDelElem subPrjLeft
    rRight = (subOutLs ** (subOutHs, rPrjRight))
    rLeft = ((l2,t) :: subInLs ** (
      (::) {l=l2} val subInHs, rPrjLeft))
   in
     (rLeft, rRight)
```

Si pertenece a la lista, entonces se utiliza la siguiente funcion para sacar esa etiqueta de las que se quieren proyectar

```
deleteElemPred : {x : t} -> (xs : List t) ->
  (elem : Elem x xs) ->
  (res : List t ** DeleteElemPred xs elem res)
```

Al llamar a `deleteElemPred ls l2InLs` se obtiene el predicado correspondiente a tal eliminacion. Luego se llama recursivamente a hProjectByLabelsHList con esa nueva lista en `hProjectByLabelsHList {lty=lty} {ts=ts2}`

`lsNew hs`. Con `IPR_ProjLabelElem {l=l2} {ty=t} {ts=ts2} {res1=subOutLs} l2InLs isDelElem subPrjRight` se genera la prueba de `IsProjectRight` correspondiente, y con `IPL_ProjLabelElem {l=l2} {ty=t} {ts=ts2} {res1=subInLs} l2InLs isDelElem subPrjLeft` la prueba de `IsProjectLeft`. Luego se retorna en la derecha `(subOutLs ** (subOutHs, rPrjRight))` que no contiene el valor actual y `((l2,t) :: subInLs ** ((::) {l=l2} val subInHs, rPrjLeft))` que si contiene el valor actual (de nuevo se utiliza el constructor de forma prefija para asignar las variables implicitas correspondientes).

```
No notL2InLs =>
  let
    ((subInLs ** (subInHs, subPrjLeft)),
      (subOutLs ** (subOutHs, subPrjRight))) =
      hProjectByLabelsHList {lty=lty} {ts=ts2} ls hs
    rPrjLeft = IPL_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2}
      {res1=subInLs} notL2InLs subPrjLeft
    rPrjRight = IPR_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2}
      {res1=subOutLs} notL2InLs subPrjRight
    rLeft = (subInLs ** (subInHs, rPrjLeft))
    rRight = ((l2,t) :: subOutLs ** (
      (::) {l=l2} val subOutHs, rPrjRight))
```

Si no pertenece a la lista, entonces simplemente se realiza la proyeccion sobre el resto de los campos y se agrega ese valor a la derecha. La proyeccion se realiza con `hProjectByLabelsHList {lty=lty} {ts=ts2} ls hs`, la prueba de `IsProjectLeft` se genera con `IPL_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2} {res1=subInLs} notL2InLs subPrjLeft`, y la prueba de `IsProjectRight` se genera con `IPR_ProjLabelNotElem {l=l2} {ty=t} {ts=ts2} {res1=subOutLs} notL2InLs subPrjRight`. El valor izquierdo devuelto es `(subInLs ** (subInHs, rPrjLeft))` y el derecho `((l2,t) :: subOutLs ** ((::) {l=l2} val subOutHs, rPrjRight))`, donde se agrega el valor actual.

Esta funcion permite proyectar sobre listas heterogeneas, pero se necesita proyectar sobre records extensibles. Para ello es necesario tener una garantia de que las etiquetas resultantes de la proyeccion no sean repetidas. Eso se logra con el siguiente lema:

```
hProjectByLabelsLeftIsSet_Lemma2 : DecEq lty =>
  {ls : List lty} -> {ts1, ts2 : LabelList lty} ->
  IsProjectLeft ls ts1 ts2 -> IsLabelSet ts1 ->
  IsLabelSet ts2
```

Este lema indica que si las etiquetas de `ts1` no son repetidas, representadas con una prueba de `IsLabelSet ts1`, y se realiza la proyeccion con `IsProjectLeft ls ts1 ts2`, entonces las etiquetas resultantes no son repetidas, representadas con una prueba de `IsLabelSet ts2`.

Con este lema se tienen todas las herramientas para definir la proyeccion sobre un Record

```
hProjectByLabelsWithPred : DecEq lty =>
```

```
    {ts1 : LabelList lty} -> (ls : List lty) ->
    Record ts1 -> IsSet ls ->
    (ts2 : LabelList lty **
      (Record ts2, IsProjectLeft ls ts1 ts2))
  hProjectByLabelsWithPred ls rec lsIsSet =
    let
      isLabelSet = recLblIsSet rec
      hs = recToHList rec
      (lsRes ** (hsRes, prjLeftRes)) =
        fst $ hProjectByLabelsHList ls hs
      isLabelSetRes = hProjectByLabelsLeftIsSet_Lemma2
        prjLeftRes isLabelSet
    in (lsRes **
      (hListToRec {prf=isLabelSetRes} hsRes, prjLeftRes))
```

El tipo `hProjectByLabelsWithPred : DecEq lty =>{ts1 : LabelList lty} ->(ls : List lty) ->Record ts1 ->IsSet ls ->(ts2 : LabelList lty ** (Record ts2, IsProjectLeft ls ts1 ts2))` toma un record `Record ts1`, una lista de etiquetas `ls` no repetidas (con una prueba de `IsSet ls`), y retorna un nuevo record `Record ts2` con una prueba de que `ts2` es una proyeccion, con `IsProjectLeft ls ts1 ts2`.

La implementacion toma la lista heterogenea del record y llama a `hProjectByLabelsHList ls hs` obteniendo el nuevo HList y la prueba `IsProjectLeft ls ts1 ts2`. Sabiendo que `rec` es un record se sabe que `IsLabelSet ts1`, por lo que, utilizando el lema `hProjectByLabelsLeftIsSet_Lemma2`, se sabe que `IsLabelSet ts2`. Con esos dos datos es posible crear el nuevo record `Record ts2`.

Para no tener que pasar una prueba manual de `IsSet ls` se define la siguiente funcion, utilizando el mismo truco definido anteriormente.

```
hProjectByLabelsWithPredAuto : DecEq lty =>
  {ts1 : LabelList lty} -> (ls : List lty) ->
  Record ts1 ->  TypeOrUnit (isSet ls)
  ((ts2 : LabelList lty **
    (Record ts2, IsProjectLeft ls ts1 ts2)))
hProjectByLabelsWithPredAuto ls rec =
  mkTypeOrUnit (isSet ls)
  (\isSet => hProjectByLabelsWithPred ls rec isSet)
```

Esta funcion utiliza el tipo `TypeOrUnit` para encapsular un predicado decidible que se debe calcular en tiempo de compilacion. A diferencia de `RecordOrUnit`, cuyo predicado era `IsLabelSet ts`, en este caso el predicado a calcular es `IsSet ls`.

`TypeOrUnit (isSet ls) rest` esconde la necesidad de calcular `isSet ls : IsSet ls` a mano y le permite a `rest` utilizar esa prueba.

3.4.1. Computacion de funcion en el tipo del record

Esta definicion es suficiente para realizar la proyeccion izquierda de una lista de etiquetas, pero resulta dificil de utilizar en la practica. Esta funcion retorna una lista de etiquetas arbitraria, el record resultante y una prueba de que es la proyeccion,

pero no retorna ninguna otra informacion que permita a uno utilizar ese record directamente.

Su uso seria el siguiente

```
rec : Record [("Nombre", String), ("Edad", Nat),
              ("Apellido", String)]
rec = consRecAuto "Nombre" "Juan" $
              consRecAuto "Edad" 24 $
              consRecAuto "Apellido" "Sanchez" $
              emptyRec

projRec : (ts2 : LabelList String **
  (Record ts2, IsProjectLeft ["Apellido", "Edad"]
  [("Nombre", String), ("Edad", Nat), ("Apellido", String)]
  ts2))
projRec = hProjectByLabelsWithPredAuto
  ["Apellido", "Edad"] rec
```

Al tener el valor `projRec` no es posible conocer el valor de `ts2` en tiempo de compilacion, cuando pareceria obvio que es `[('Apellido', String), ('Edad', Nat)]`. Para tener informacion sobre esa lista es necesario hacer un analisis de casos sobre la prueba de `IsProjectLeft`.

Esto resulta util cuando se utiliza en otras funciones que necesitan realizar ese analisis de casos, o tienen en su alcance o contexto otras pruebas y tipos que pueden hacer uso de ese predicado. Pero para la mayoria de sus usos quisieramos utilizarlo de la forma que se indico al comienzo de esta seccion

```
projectedRec : Record [("Apellido", String),
  ("Edad", Nat)]
projectedRec = hProjectByLabelsAuto
  ["Apellido", "Edad"] rec
```

Para ello es necesario computar la lista resultante en el tipo mismo del record. Es decir, se necesita que en tiempo de compilacion la funcion de proyeccion tome la lista de etiquetas del record, la lista de etiquetas que se quieren proyectar, y en el tipo del record compute tal proyeccion y retorne la lista resultante.

Esto es posible gracias a estas dos funciones definidas anteriormente

```
projectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty

fromIsProjectLeftToComp : DecEq lty =>
  {ls : List lty} -> {ts1, ts2 : LabelList lty} ->
  IsProjectLeft ls ts1 ts2 ->
  IsSet ls -> ts2 = projectLeft ls ts1
```

`projectLeft` permite computar la lista resultante en el tipo, y `fromIsProjectLeftToComp` permite tomar una prueba del predicado `IsProjectLeft ls ts1 ts2` y asegurar que su resultado sea identico a la computacion `projectLeft ls ts1`. Con estas funciones se puede definir la siguiente funcion

```
hProjectByLabels : DecEq lty =>
  {ts : LabelList lty} -> (ls : List lty) ->
  Record ts -> IsSet ls ->
  Record (projectLeft ls ts)
hProjectByLabels {ts} ls rec lsIsSet =
  let
    isLabelSet = recLblIsSet rec
    hs = recToHList rec
    (lsRes ** (hsRes, prjLeftRes)) =
      fst $ hProjectByLabelsHList ls hs
    isLabelSetRes = hProjectByLabelsLeftIsSet_Lemma2
      prjLeftRes isLabelSet
    resIsProjComp =
      fromIsProjectLeftToComp prjLeftRes lsIsSet
    recRes = hListToRec {prf=isLabelSetRes} hsRes
  in rewrite (sym resIsProjComp) in recRes
```

La funcion es similar a `hProjectByLabelsWithPred`, pero en vez de retornar `(ts2 : LabelList lty ** (Record ts2, IsProjectLeft ls ts1 ts2))` realiza la computacion de la lista resultado en el tipo mismo y la retorna en `Record (projectLeft ls ts)`.

Su implementacion es identica a `hProjectByLabelsWithPred`, realizando la proyeccion sobre la lista heterogenea con `hProjectByLabelsHList` y obteniendo una prueba de etiquetas no repetidas con `hProjectByLabelsLeftIsSet_Lemma2`. Sin embargo, como la proyeccion sobre HList retorno una prueba de `IsProjectLeft ls ts ts2` en `prjLeftRes`, se puede llamar a `fromIsProjectLeftToComp prjLeftRes lsIsSet` para tener la igualdad `ts2 = projectLeft (ls ts)`. La llamada a `hListToRec {prf=isLabelSetRes} hsRes` retorna un record utilizando esa misma lista `ts2` (ya que `isLabelSetRes` es una prueba de `IsLabelSet ts2`). Por lo tanto, teniendo esa igualdad se puede utilizar la herramienta `rewrite (sym resIsProjComp) in recRes` para modificar el tipo de `recRes` de `Record ts2` a `Record (projectLeft ls ts)`.

`rewrite proof in rest` toma una prueba de igualdad `proof` (`projectLeft (ls ts) = ts2` en este caso) y realiza una sustitucion del termino de la derecha con el de la izquierda en `rest` (`recRes` en este caso).

Siguiendo con la estrategia de `hProjectByLabelsWithPredAuto`, se define otra funcion que evita tener que calcular `IsSet ls` a mano

```
hProjectByLabelsAuto : DecEq lty =>
  {ts : LabelList lty} -> (ls : List lty) ->
  Record ts -> TypeOrUnit (isSet ls)
  (Record (projectLeft ls ts))
hProjectByLabelsAuto {ts} ls rec =
  mkTypeOrUnit (isSet ls)
  (\lsIsSet => hProjectByLabels {ts=ts} ls rec lsIsSet)
```

Esta implementacion es identica a `hProjectByLabelsWithPredAuto`.

Con estas nuevas funciones, por fin es posible realizar la proyeccion de la siguiente manera

```
projectedRec: Record [("Apellido", String),
  ("Edad", Nat)]
projectedRec = hProjectByLabelsAuto
  ["Apellido", "Edad"] rec
```

3.5. 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 continuacion se describen tales formas, indicando por que no se opto por cada una ellas.

3.5.1. 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 informacion de ellos en su tipo. Cada valor es simplemente reconocido como `HValue`, y no es posible conocer su tipo u operar con el 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 utilizo este enfoque ya que al no poder obtenerse la informacion del tipo de `HValue` es imposible poder verificar que un record contenga campos con etiquetas y que estos no esten 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
```

3.5.2. 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 definicion 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 informacion 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
```

3.5.3. 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 definicion es un punto medio (en terminos de poder) entre la definicion 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 funcion que toma un tipo y construye otro tipo a partir de este. Esta definicion 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 informacion 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 definicion 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).