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. Surgen como una respuesta a un problema que tienen los 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 primitiva de definir y manipular records extensibles. Algunos lenguajes ni siquiera permiten definirlos.

Este trabajo se enfoca en presentar una manera de definir records extensibles en lenguajes funcionales fuertemente tipados. En particular, se presenta una manera de hacerlo utilizando un lenguaje de programación con *tipos dependientes* llamado Idris.

1.1. Records Extensibles

En varios lenguajes de programación con tipos estáticos, es posible definir una estructura estática llamada 'record'. Un record es una estructura heterogénea que permite agrupar valores de varios tipos en un único objeto, asociando una etiqueta o nombre a cada uno de esos valores. Un ejemplo de la definición de un tipo record en Haskell sería la siguiente:

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

En muchos lenguajes de programación, 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.

```
data Persona2 = Persona2 {
  edad :: Int,
  nombre :: String,
  apellido: String
}
```

- Crear un nuevo record que contenga el campo `apellido` y otro campo de tipo `Persona`.

```
data Persona2 = Persona2 {
  datosViejos: Persona,
  apellido : String
}
```

En ninguno de ambos enfoques es posible obtener el nuevo record extendiendo el anterior 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 este 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 tipo `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 implementar 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.

Algunos lenguajes permiten implementar records extensibles con primitivas del lenguaje (como en Elm [9] o Purescript [23]), pero requieren de sintaxis y semántica especial para records extensibles, y no es posible implementarlos con primitivas del lenguaje más básicas.

Las propiedades de records extensibles requieren que los campos sean variables y puedan ser agregados a records previamente definidos. En el ejemplo visto más arriba, básicamente se espera poder tener un tipo `Persona`, y poder agregarle un campo `apellido` con el tipo `String`, y que esto sea un nuevo tipo. Esto se puede realizar con *tipos dependientes*.

1.2. Tipos Dependientes

Tipos dependientes son tipos que dependen de valores, y no solamente de otros tipos. En el ejemplo anterior, el nuevo tipo `Persona + { apellido :: String}` depende de tipos (`Persona` y `String`) pero también de valores. El elemento `apellido` se puede representar como un valor de tipo `String`, quedando el tipo final `Persona + { 'apellido' :: String }`. Con tipos dependientes los tipos son *first-class*, lo que permite que cualquier operación que se puede

realizar sobre valores también se puede realizar sobre tipos. En particular, se pueden definir funciones que tengan tipos como parámetros y retornen tipos como resultado. En el ejemplo, + funciona como tal función, tomando el tipo `Persona` y el tipo `{'apellido' :: String}` y uniéndolos en un nuevo tipo. `::` funciona como otra función, uniendo el valor `apellido` y el tipo `String`.

En relacion 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 informacion 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.

Otra propiedad de los tipos dependientes es que permiten definir propiedades como tipos. Es posible definir proposiciones como tipos, y probar tales proposiciones construyendo valores de ese tipo. Como ejemplo, se puede codificar la propiedad *'Un record extensible no puede tener campos repetidos'* como un tipo `RecordSinRepetidos record` (donde `record` es un valor de tipo record), y luego se puede usar un valor `prueba : RecordSinRepetidos record` en cualquier lugar donde uno quiera que se cumpla esa propiedad. Como se verá más adelante en el trabajo, esto permite tener chequeos y restricciones sobre la construcción de records en tiempo de compilación, ya que si un tipo que representa una propiedad no puede ser construido (lo que significa que no se pudo probar esa propiedad), entonces el código no va a compilar. Esto es una mejora a la alternativa de que se realice un chequeo en tiempo de ejecución haciendo que falle el programa, ya que uno no necesita correr el programa para saber que éste es correcto y cumple tal propiedad.

1.3. Idris

En este trabajo se decidio utilizar el lenguaje de programacion *Idris* [3] para llevar a cabo la investigacion del uso de tipos dependientes aplicados a la definicion de records extensibles. Idris es un lenguaje de programacion funcional con tipos dependientes, con sintaxis similar a Haskell.

Otro lenguaje que cumple con los requisitos es *Agda* [22], un 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.

1.4. Guía de trabajo

Este trabajo se divide en varias secciones.

En el capítulo *Estado del arte* se van a mostrar varias implementaciones de records extensibles en otros lenguajes, mostrando sus beneficios e inconvenientes. Eventualmente se enfocará en una implementación específica de records extensibles en Haskell.

En el capítulo *Records Extensibles en Idris* se presentará la implementación de records extensibles en Idris llevada a cabo en este proyecto, expandiendo en el uso de Idris y las funcionalidades de éste que hicieron posible este trabajo, y mostrando los problemas encontrados y sus respectivas soluciones.

En el capítulo *Caso de Estudio* se presentará un caso de estudio de construcción y evaluación de expresiones aritméticas, haciendo uso de lo desarrollado en este trabajo, permitiendo ver cómo es el uso de esta implementación y qué se puede llegar a hacer con ella.

En el capítulo *Conclusión* se tomarán las experiencias del desarrollo del trabajo y el caso de estudio, y se analizarán los problemas encontrados, el aporte que puede llegar a tener este trabajo sobre el problema de records extensibles, etc.

En el capítulo *Trabajo a futuro* se indicarán aspectos de diseño y implementación que pueden realizarse sobre este trabajo para mejorarlo, como posibles tareas que surgieron y se pueden llevar a cabo.

En el *Apéndice* se encontrarán recursos adicionales, como el código fuente de este trabajo.

Capítulo 2

Estado del arte

En este capítulo se van a presentar varias implementaciones de records extensibles en varios lenguajes. Se enfocará en lenguajes fuertemente tipados. Existen implementaciones de records extensibles en lenguajes dinámicamente tipados, pero no entran en el alcance de este trabajo ni de lo que se quiere investigar.

En términos generales, las implementaciones de records extensibles se dividen según si son proporcionados por el lenguaje como primitivas, o si son proporcionados por bibliotecas. Se describirán ambas alternativas, incluyendo ejemplos de lenguajes e implementaciones de cada uno.

En particular, este trabajo se enmarca dentro de los proporcionados como bibliotecas de usuario. Este trabajo fue motivado por una biblioteca de Haskell llamada *HList*, la cual va a ser presentada más adelante.

2.1. Records Extensibles como primitivas del lenguaje

Algunos lenguaje de programación funcionales permiten el manejo de records extensibles como primitiva del lenguaje. Esto significa que records extensibles son funcionalidades del lenguaje en sí, y tienen sintaxis y funcionamiento especial en el lenguaje.

Uno de ellos es *Elm*[6]. Uno de los ejemplos que se muestra en su documentación ([9]) 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 (o *rows*, como es descrito en la literatura). 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 [8].

Otro lenguaje con esta particularidad es *Purescript*[24]. A continuación se muestra uno de los ejemplos de su documentación [23]:

```
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 *Extensible records with scoped labels* [20], de Daan Leijen.

También existen otras propuestas de sistemas de tipos con soporte para records extensibles. En *First-class labels for extensible rows*[21], Daan Leijen describe un sistema de tipos con etiquetas como *first-class citizens* del lenguaje para permitir construir records extensibles de forma expresiva. En *A Polymorphic Type System for Extensible Records and Variants* [12], Benedict R. Gaster y Mark P. Jones también describen un sistema de tipos con soporte para records extensibles. Ambos se basan en extensiones del sistema de tipos de Haskell y ML, incluyendo inferencia de tipos. Este paper de Gaster y Jones fue utilizado para la extensión Trex del compilador Hugs98 [1]. Hugs98 es un compilador de Haskell, cuya extensión Trex soporta records extensibles utilizando los fundamentos teóricos del paper de Gaster y Jones. Desafortunadamente, Hugs98 no está más en desarrollo. Otra propuesta es la de *Operations on Records*[4], de Luca Cardelli y John C. Mitchell. Esta propuesta crea una teoría de records extensibles, formalizando un sistema de tipos para soportarlos. Otra propuesta es *Lightweight Extensible Records for Haskell*[16], de Mark P. Jones

y Simon Peyton Jones, la cual describe una extensión de Haskell98 para soportar records extensibles. Muchas de las propuestas son similares y utilizan los mismos fundamentos de la teoría de records.

Estas propuestas, al igual que cualquier propuesta de extender un lenguaje para que soporte extensible records como primitivas, tienen algunas desventajas. La desventaja principal es que se deben agregar nuevas reglas de tipado para soportar los records, y esto puede impactar otros aspectos del lenguaje. En el paper *Lightweight Extensible Records for Haskell*[16], los autores describen un problema de su extensión de records, donde al tener un nuevo kind (tipo de tipos) sobre records, existen ambiguedades y problemas al integrarlo con el sistema de *typeclasses* de Haskell. Al agregar una nueva regla al lenguaje para soportar records, esta regla puede llegar a ser inconsistente con otras partes del lenguaje, y puede requirir un rediseño de muchas partes del lenguaje para que funcionen correctamente con estas nuevas reglas. Esto también genera que el lenguaje sea más costoso de mantener y entender, lo cual puede traer problemas futuros cuando se intenten implementar nuevas funcionalidades y extensiones del lenguaje. Otro problema es que, al menos que se explicite, las nuevas funcionalidades del lenguaje que soportan records extensibles no son *first-class*. 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 el usuario mismo.

Este trabajo se enfocará en records extensibles como bibliotecas de usuario. Una biblioteca no modifica el lenguaje mismo sino que hace uso de todas sus funcionalidades ya definidas, por lo que no presentan los problemas vistos anteriormente. A su vez las bibliotecas le dan varias opciones al usuario, que puede elegir la implementación de records extensibles que desea y le resulta mejor para su situación. Otra ventaja es que poder definir records extensibles como una biblioteca evita que uno necesite cambiar el lenguaje para poder soportarlos.

2.2. Records Extensibles como bibliotecas de usuario

Las bibliotecas de usuario son componentes de un lenguaje de programación escrito en ese lenguaje para proveer una funcionalidad al usuario. Las implementaciones de records extensibles que utilizan este mecanismo se basan en utilizar funcionalidades avanzadas del lenguaje en cuestión para poder realizar la definición de tales records.

En *Generic Record Combinators with Static Type Checking* [14], Wolfgang Jeltsch describe un sistema de records en Haskell definiendo los records como typeclasses, y los tipos y etiquetas del record como tipos producto. Su trabajo fue liberado como una biblioteca de Haskell llamada *records* [15]. Esta implementación es similar a HList, pero no nos enfocaremos en ella en este trabajo.

Otras propuestas se basan en utilizar extensiones de Haskell para poder definir los records. *rawr* [5] utiliza *type families* y *type-level lists* para definir sus records. *Vinyl* [25] también utiliza type-level lists, pero define el record como un GADT (Generalized Abstract Data Type). *ruin* [11] utiliza *Template Haskell* y metaprogramming para crear instancias de type-classes específicas. Existen otras propuestas, como *labels* [7], *named-records* [10], *bookkeeper* [2], entre otras. Estas bibliotecas son muy similares en su funcionamiento, difiriendo en qué funcionalidades de GHC (Glasgow Haskell

Compiler) y Haskell utilizan, qué funcionalidades de records y records extensibles le proporcionan al usuario, qué sintaxis usan, etc.

Como ejemplo de uso de una biblioteca, se tienen los records de *rawr* que se definen de la siguiente forma:

```
type Foo = R ( "a" := Int, "b" := Bool )
let foo = R ( #a := 42, #b := True ) :: Foo
```

La definición de un record se realiza llamando a funciones y tipos proporcionados por la biblioteca (como es el caso de las funciones `R` en este ejemplo). Para realizar la extensión de un record, generalmente también se utilizan operadores definidos en la biblioteca, como en el siguiente caso:

```
R ( #foo := True ) :*: R ( #bar := False )
--> R ( bar := False, foo := True )
```

Existen muchas alternativas y propuestas, pero el trabajo actual se basó en la biblioteca HList de Haskell. HList muestra las problemáticas de definir records extensibles, a la vez que muestra soluciones a ellas. También tiene varias propiedades que son deseadas tener en records extensibles (como flexibilidad para agregar nuevas operaciones sobre records), y el marco de este trabajo es transladarlo a un contexto de tipo dependientes en Idris y hacer uso de esas propiedades.

Uno de los mecanismos utilizados por HList y la mayoría de las bibliotecas previamente descritas son las *listas heterogéneas*, que se presentarán más adelante. Bibliotecas como HList y Records (de Wolfgang Jeltsch) son más viejas y utilizan funcionalidades más básicas de Haskell para definir listas heterogéneas (como type families y type classes). Algunas de las bibliotecas mencionadas (como *vinyl*) son más modernas y utilizan funcionalidades nuevas de GHC, como `DataKinds`, que permite definir listas heterogéneas a nivel de tipos, y así permitir definir records extensibles de forma más sencilla.

Las listas heterogéneas son un pilar base de muchas implementaciones de records extensibles, pero en particular son fundamentales para la implementación de HList, y subsecuentemente para la implementación de records extensibles en Idris. A continuación se describirán las listas heterogéneas, luego se mostrará cómo se implementan en la biblioteca HList de Haskell, y luego se presentará cómo son implementadas en Idris.

2.3. 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.

2.3.1. HList en Haskell

Para definir listas heterogéneas en Haskell nos basaremos en la propuesta presentada en *Strongly Typed Heterogeneous Collections* [18], por Oleg Kiselyov, Ralf Lämmel y Keean Schupke. Esta propuesta está implementada en el paquete *hlist*, encontrado en Hackage [17].

Esta biblioteca 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`.

Un ejemplo de uso de tal función sería el siguiente:

```
HLength '[Int, String] = HSucc (HSucc HZero)
```

2.3.2. 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` (este tipo se verá a continuación).

Para conocer mejor el uso de tipos dependientes en Idris, veamos la definición del tipo *vector* que son listas cuyo largo es anotado 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` es una lista que contiene 1 string, mientras que un valor del tipo `Vect 10 String` es una lista que contiene 10 strings. En la definición `Vect : Nat ->Type ->Type` el tipo mismo queda parametrizado por un natural además de un tipo cualquiera.

Tener valores en el tipo permite poder restringir el uso de funciones a determinados tipos que tengan valores específicos. Como ejemplo se tiene la siguiente funcion:

```
head : Vect (n + 1) a -> a
head (x :: xs) = x
```

Esta funcion obtiene el primer valor de un vector. Es una funcion total, ya que restringe su uso solamente a vectores que tengan un largo mayor a 0, por lo que tal vector siempre va a tener por lo menos un elemento para obtener. Si se intenta llamar a esta función con un vector sin elementos, como `head []`, la llamada no

va a compilar porque el type-checker no va a saber unificar `Vect 0 a` con `Vect (n + 1) a` (no puede encontrar un valor natural `n` que cumpla `n + 1 = 0`, por lo que falla el type-checking).

En cuanto a listas heterogéneas, la definición de HList utilizada en Idris es la siguiente:

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

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

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

El tipo `HList` se define como una función de tipo que toma una lista de tipos (e.j `[Nat, String]`) 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.

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 bibliotecas y los usuarios de tales, disminuyendo la dificultad de su uso para resolver problemas.

A continuación se presentará cómo la biblioteca de Haskell HList define records extensibles utilizando estas listas heterogéneas.

2.4. Records Extensibles en Haskell

La propuesta de HList [18] utiliza listas heterogéneas para definir un record:

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

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

```
mkRecord = Record
```

Un record se representa simplemente como una lista heterogénea de un tipo determinado (que se verá más adelante). Un record se puede construir solamente utilizando `mkRecord`. Esta función toma una lista heterogénea, pero fuerza a que tenga una instancia de `HRLabelSet`

Una lista heterogénea con una instancia para esta type-class implica que la lista tiene valores con etiquetas, y ninguna etiqueta se repite. Se define de la siguiente forma:

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

Para poder implementar una instancia de esta type-class, la lista necesita cumplir el predicado `HAllTaggedLV` y `HLabelSet`. Para esto la lista debe contener este tipo:

```
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.

El predicado `HAllTaggedLV` simplemente verifica que la lista solo contenga elementos del tipo `Tagged`. Ambos `Tagged` y `HAllTaggedLV` pertenecen a la biblioteca *tagged* [19].

Un ejemplo posible de tal lista sería

```
Tagged 10 `HCons` Tagged 'John' `HCons` HNil ::
  HList '[Tagged s1 Nat, Tagged s2 String]
```

El valor en sí no contiene las etiquetas `s1` ni `s2`, pero el tipo las contiene. Estas etiquetas no deben repetirse, y para eso la lista debe poder tener una instancia de `HLabelSet (LabelsOf ps)`. `LabelsOf` es una type-family que toma una lista de `Tagged` y obtiene sus etiquetas:

```
type family LabelsOf (ls :: [*]) :: [*]
type instance LabelsOf '[] = '[]
type instance LabelsOf (Label l ': r)  = Label l ': LabelsOf r
type instance LabelsOf (Tagged l v ': r) = Label l ': LabelsOf r
```

`LabelsOf ls` toma todos los *phantom type* de `Tagged` y los retorna en una lista a nivel de tipos. Como en algunas partes de la implementación se permite que la lista heterogénea tenga solo etiquetas (sin valores) tal lista puede tener un valor del tipo `Label l`. Este tipo `Label` simplemente permite construir una etiqueta a nivel de tipos para poder identificar los campos del record.

`HLabelSet` es una type-class que representa el predicado de que las etiquetas de la lista no estén repetidas. Para ello se aplica la función `LabelsOf` antes, para poder obtener solo las etiquetas de la lista (y no sus valores). Este predicado se define de forma recursiva definiendo instancias para cada caso base y caso inductivo. Hace uso de predicados de igualdad de tipos para poder realizarlo. No se mostrará su implementación.

La biblioteca también proporciona otra forma de generar records utilizando etiquetas y operadores especiales. Un ejemplo de su uso es:

```
data PersonaNamespace = PersonaNamespace
clave = firstLabel PersonaNamespace "clave"
nombre = nextLabel clave "nombre"
edad = nextLabel nombre "edad"

persona = clave .=. (3 :: Integer)
  .*. nombre .=. 'Juan'
  .*. edad .=. 27
  .*. emptyRecord
```

Funciones como `firstLabel` y `nextLabel` permiten construir elementos del tipo `Label t`, que luego pueden usarse para definir los campos del record con el siguiente operador:

```
(.=.) :: Label l -> v -> Tagged l v
```

El operador `(.*.)` permite extender un record con un nuevo campo. Su implementación no se mostrará, pero cabe notar que su implementación termina realizando un llamado a la función `mkRecord` definida anteriormente.

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
```

Para este trabajo se tomó como motivación esta implementación de records, realizando una traducción a Idris de cada implementación, cada tipo y cada algoritmo. A continuación se mostrará la implementación realizada en este trabajo.

Capítulo 3

Records Extensibles en Idris

En este capítulo se describiá cómo se implementaron los records extensibles en Idris. Se comenzará mostrando ejemplos de creación y uso de records. Luego se explicará el diseño de los records y cuáles funcionalidades de Idris lo hicieron posible. En otra sección se mostrará la implementación de algunas de las funciones sobre records extensibles. No se mostrarán sus implementaciones en su completitud, sino que se describirán los rasgos más importantes de la implementación, dejando los detalles y funciones auxiliares para ver en el apéndice. El capítulo terminará comparando esta solución de records extensibles con otras vistas en el capítulo *'Estado del Arte'*.

3.1. Primer pantallazo de records extensibles en Idris

En este trabajo se decidió seguir el diseño de HList de Haskell para extender records. Como ejemplo, podemos tomar el siguiente caso de HList descrito en la sección anterior:

```
persona = clave .=. (3 :: Integer)
  .*. nombre .=. 'Juan'
  .*. edad .=. 27
  .*. emptyRecord
```

Este ejemplo se puede definir en Idris de esta forma:

```
persona : Record [('Clave', Nat), ('Nombre', String),
  ('Edad', Nat)]
persona = consRecAuto 'Clave' 3 $
  consRecAuto 'Nombre' 'Juan' $
  consRecAuto 'Edad' 27 $
  emptyRec
```

Las funciones y valores utilizados son comparables con las de HList. La única diferencia entre ambos ejemplos (además de leves diferencias sintácticas) es en la declaración de tipo de Idris.

```
persona : Record [('Clave', Nat), ('Nombre', String),
  ('Edad', Nat)]
```

El tipo base de este trabajo es `Record`. Como se ve, `Record` contiene en su tipo la lista `[('Clave', Nat), ('Nombre', String), ('Edad', Nat)]`. Esta lista representa los campos (o *rows*) del record. Es una lista de tuplas, donde el primer valor de la tupla es la etiqueta del campo, y el segundo valor es el tipo del campo. Aquí está el uso base de tipos dependientes de este trabajo, ya que el tipo `Record` depende de un valor (una lista en particular).

En Idris el tipo de Record sería `Record : List (String, Type) ->Type`. Representa una función de tipo, que se puede aplicar a un valor y retornar un tipo nuevo como `Record [('Clave', Nat), ('Nombre', String), ('Edad', Nat)] : Type`.

Dentro del tipo del record se encuentra toda la información necesaria, ya que se encuentran todas las etiquetas del mismo y el tipo de cada campo. Tampoco es necesario definir valores externos al record como las etiquetas `clave` de HList (de tipo `Label`), sino que se pueden usar simples tipos como `String` para definir las etiquetas.

Otra pieza fundamental en este trabajo es `consRecAuto`. Esta es la función que permite tomar un record y extenderlo con otro valor. Para poder entender su comportamiento se puede ver el mismo ejemplo pero con menos campos:

```
persona2 : Record [('Nombre', String), ('Edad', Nat)]
persona2 = consRecAuto 'Nombre' 'Juan' $
  consRecAuto 'Edad' 27 $
  emptyRec
```

Al tener este nuevo record, se puede reescribir el original de esta forma:

```
persona1 : Record [('Clave', Nat), ('Nombre', String),
  ('Edad', Nat)]
persona1 = consRecAuto 'Clave' 3 persona2
```

Si comparamos estos dos records, podemos ver que `persona1` contiene la tupla `('Clave', Nat)` en su tipo mientras que `persona2` no. Al comparar los argumentos pasados a `consRecAuto` y cómo éste modificó el tipo del record resultante, se puede definir su tipo de esta forma:

```
consRecAuto : {A : Type} -> (l : String) -> (a : A) ->
  Record ts -> Record ((l, A) :: ts)
```

En Idris los paréntesis `{}` indican argumentos implícitos mientras que `()` indican argumentos explícitos. El compilador intenta inferir los argumentos implícitos conociendo cuales otros argumentos fueron pasados a la función, aunque a veces no puede y es necesario pasarlos explícitamente. `consRecAuto 'Clave' 3 persona2` es equivalente a `consRecAuto {A = Nat} 'Clave' 3 persona2`.

Inductivamente se puede seguir aplicando el razonamiento anterior para conocer el tipo de `emptyRec`

```
emptyRec : Record []
```

Este trabajo también permite realizar varias operaciones de creación, manipulación y lookup de records.

```
personaConNombre : Record [("Clave", Nat), ("Nombre", String)]
personaConNombre = consRecAuto "Clave" 1 $
  consRecAuto "Nombre" "Juan" $
  emptyRec
-- { "Clave": 1, "Nombre": "Juan" }

personaConEdad : Record [("Clave", Nat), ("Edad", Nat)]
personaConEdad = consRecAuto "Clave" 1 $
  consRecAuto "Edad" 34 $
  emptyRec
-- { "Clave": 1, "Edad": 34 }
```

Si uno tiene estos dos records que representan distintos atributos de una persona, entonces puede unirlos de la siguiente manera:

```
persona : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat)]
persona = hLeftUnion personaConNombre personaConEdad
-- { "Clave": 1, "Nombre": "Juan", "Edad": 34 }
```

Esta unión unifica los campos de ambos (quedándose con el valor del record de la izquierda para los campos repetidos).

Uno también puede obtener el valor de cualquiera de sus campos:

```
nombre : String
nombre = hLookupByLabelAuto "Nombre" persona
-- "Juan"
```

Si se quiere también se puede actualizar un campo individualmente:

```
personaActualizada : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat)]
personaActualizada = hUpdateAtLabelAuto "Nombre" "Pedro" persona
-- { "Clave": 1, "Nombre": "Pedro", "Edad": 34 }
```

Si uno tiene un record con valores completamente distintos, también puede simplemente añadirlos al final del record original:

```
direccion : Record [("Direccion", String)]
direccion = consRecAuto "Direccion" "18 de Julio" $
  emptyRec

personaYDireccion : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat), ("Direccion", String)]
personaYDireccion hAppendAuto persona direccion
-- { "Clave": 1, "Nombre": "Juan", "Edad": 34,
  "Direccion": "18 de Julio" }
```

Si se tiene un record, se puede obtener un sub-record proyectando por algunos de sus campos:

```
proyeccion: Record [("Clave", Nat), ("Direccion", String)]
proyeccion = hProjectByLabelsAuto ["Clave", "Direccion"]
  personaYDireccion
-- { "Clave": 1, "Direccion": "18 de Julio"}
```

Todas estas funciones fueron traducidas de la biblioteca HList de Haskell. La implementacion y descripcion de ellas se vera mas adelante.

3.2. Predicados y propiedades en Idris

Antes de ver la implementacion de `consRec` y `emptyRec`, es necesario entender la otra propiedad de estos records. Al igual que en HList, se desea que en tiempo de compilación se sepa que las etiquetas del record no son repetidas. En particular, para `consRecAuto`, se necesita saber que la etiqueta nueva a agregar no existe actualmente en el record. Por lo tanto, se necesita más información del record y del campo a agregar para poder extender un record.

La función que extiende un record debería tener el siguiente tipo:

```
consRec : {A : Type} -> (l : String) -> (a : A) ->
 (prf : Not (Elem l ts)) -> Record ts -> Record ((l, A) :: ts)
```

Se agregó el nuevo término `prf : Not (Elem l ts)`. En Idris esto representa no solo un valor `prf` de un tipo en particular, sino que representa un predicado o proposición, cuya prueba es el valor `prf` mismo. En este caso, el tipo representa la proposición *La etiqueta 'l' no pertenece a la lista de campos 'ts'*.

Esta correspodencia entre tipos y proposiciones se llama *Curry-Howard isomorphism* [13], la cual demuestra que todo predicado de una lógica constructivista puede ser representado (isomórficamente) con un tipo de un lenguaje inductivo con tipos dependientes (siempre y cuando éste sea total y consistente).

Tal predicado puede definirse como cualquier otro tipo de datos, definiendo los casos base y los casos inductivos como constructores de tal.

```
data Elem : a -> List a -> Type where
     Here : Elem x (x :: xs)
     There : Elem x xs -> Elem x (y :: xs)
```

Esta es la definición de Idris del tipo `Elem`, donde `Elem x xs` representa el predicado *El elemento 'x' se encuentra en la lista 'xs'*. La definición es inductiva, se tiene el caso base en `Here` y el caso inductivo en `There`.

El caso base ocurre cuando el elemento a comparar es idéntico al primer elemento de la lista, representado por el tipo `Elem x (x :: xs)`. Si uno quiere probar tal caso, simplemente construye el valor `Here` y lo obtiene.

El caso inductivo ocurre cuando uno sabe que el elemento está en el resto de la lista, por lo cual uno sabe que también pertenece a esa lista con cualquier otro elemento agregado a su cabeza.

Con esta definición, se puede probar un predicado simplemente construyendo términos, como en este ejemplo:

```
Here : Elem 3 [3] -- Elem 3 (3 :: [])
There Here : Elem 3 [4, 3] -- Elem 3 (4 :: 3 :: [])
There (There Here) : Elem 3 [1, 4, 3]
  -- Elem 3 (1 :: 4 :: 3 :: [])
```

En el caso anterior, se tenía el tipo `Not (Elem l ts)`. `Not` es una función de tipos que toma un tipo y retorna otro, pero representa la negación. Si tengo un valor `prf : Not (Elem l ts)`, significa que es imposible obtener cualquier prueba de `Elem l ts`, básicamente contradiciendo tal predicado.

```
Not : Type -> Type
Not t = t -> Void
```

`Not` es una simple función, que toma un tipo, y retorna un tipo función de ese tipo a `Void`. `Not (Elem l ts)` es equivalente a `Elem l ts ->Void`.

`Void` es un tipo muy interesante en Idris. Es el tipo *bottom*, es un tipo sin ninguna instancia, es un tipo que no puede construirse. En lógica constructivista, representa el valor *False*, donde si se puede obtener una prueba de ese valor, entonces se puede obtener una prueba de cualquier otro. En Idris esa regla está representada por una función proporcionada por el lenguaje llamada `absurd`

```
absurd : {a : Type} -> Void -> a
```

Si en algún momento se tiene un valor `v : Void`, entonces siempre se puede obtener cualquier tipo con `absurd v : a`, sea el tipo que sea. Esta función es generalmente usada cuando se está en un caso de pattern-matching 'imposible', y se quiere probar que es imposible de que la ejecución del programa llegue a ese caso, por lo que se prueba `Void` y luego se aplica `absurd`.

En Idris, la forma más directa de crear pruebas de `Void` es mediante la imposibilidad de aplicar constructores. Si al hacer inducción o pattern-matching sobre un valor es imposible poder encontrar un constructor que retorne un tipo compatible con él, entonces se puede utilizar el término `impossible` y crear una prueba de `Void` (básicamente, se encuentra un término que debería ser imposible de construir, por lo que se probó el absurdo).

```
noEmptyElem : Elem x [] -> Void
noEmptyElem Here impossible
```

En este ejemplo, si se tiene un valor de tipo `Elem x []`, no es posible construirlo. Si se hubiera construido con `Here`, debería tener un tipo que unifique con `Elem x (x :: xs)`, lo cual es imposible ya que la lista proporcionada es vacía. Si se hubiera construido con `There`, debería tener un tipo que unifique con `Elem x (y :: xs)`, el cual tiene el mismo problema. Por lo tanto es imposible construir un valor de tipo `Elem x []`, por lo que si se tiene tal valor, al aplicarle la función `noEmptyElem` a él se puede probar Void.

Si uno recuerda la definición de `Not` vista anteriormente, se puede reescribir esta función de esta forma:

```
noEmptyElem : Not (Elem x [])
noEmptyElem Here impossible
```

Esta forma es la forma directa de construir pruebas de `Not`.

Volviendo al caso de records extensibles, vamos a mostrar la definicion completa de la extension de records. Sin embargo, primero mostraremos algunas funciones, tipos, nomenclatura y conceptos necesarios para poder hacerlo.

En una primera instancia, describiremos que son los tipos decidibles y para que son utilizados en este trabajo.

3.2.1. Tipos Decidibles

Para poder definir records extensibles en Idris, es necesario trabajar con tipos o proposiciones 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`.

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 `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`, 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.

En este trabajo los tipos decidibles son principalmente utilizados para generalizar las etiquetas a tipos que no sean String (incluso si en este trabajo principalemente se trabaja con Strings). Su uso es bastante simple, en vez de tener `LabelList String` se tiene `DecEq lty =>LabelList lty`. Como solo se utiliza la igualdad de strings (y nada más) esta definición es suficiente. Sin embargo, a su vez es flexible para permitir otros posibles tipos de etiquetas (como enumerados finitos, identificadores naturales, entre otros).

Para poder implementar la extensión de records contamos con las siguientes funciones y tipos también:

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

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

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

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

`LabelList` es una abstracción que representa una lista de campos con etiquetas y tipos. Por ejemplo: `[('Clave', Nat)] : LabelList String`.

`labelsOf` simplemente toma una lista de campos y obtiene solo la lista con sus etiquetas. Por ejemplo: `labelsOf [('Clave', Nat), ('Edad', Nat)] = ['Clave', 'Edad']`.

`ElemLabel` toma una etiqueta, una lista de campos y representa el predicado de que esa etiqueta no pertenece a esa lista.

`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. Se basa en una función de decisión ya existente en las bibliotecas base de Idris llamada `isElem`:

```
isElem : DecEq a => (x : a) -> (xs : List a) ->
  Dec (Elem x xs)
```

Con estos conceptos y definiciones podemos llegar a la definición final de extensión de un record:

```
consRec : DecEq lty => {ts : LabelList lty} ->
  {t : Type} -> (l : lty) -> (val : t) ->
  Record ts -> {notElem : Not (ElemLabel l ts)} ->
  Record ((l, t) :: ts)
```

`DecEq lty` indica que la etiqueta debe tener igualdad (para la mayoría de los casos basta con `String`). `ts : LabelList lty` es la lista de campos del record actual, `t` es el tipo del nuevo campo, `l` es la nueva etiqueta, `val` el valor del nuevo campo, `Record ts` el record a extender, `Not (ElemLabel l ts)` la prueba de que `l` no se encuentra repetida en `ts`, y el record extendido va a tener el tipo `Record ((l, t) :: ts)`.

3.2.2. Listas sin repetidos

En la sección anterior vimos que para garantizar que no hayan etiquetas repetidas, era necesario tener una prueba de `Not (ElemLabel l ts)` (utilizando los nuevos tipos y funciones definidos anteriormente). Sin embargo, existe otra forma más sencilla de definir tal predicado. En vez de indicar que `l` no debe pertenecer a `ts`, se puede probar que la lista `(l, A) :: ts` no tiene repetidos.

Esto se define con el siguiente predicado:

```
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. 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 en este trabajo se manejan listas de campos (representados por tuplas donde el primer elemento contiene la etiqueta) se define el siguiente tipo:

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

3.2.3. Construccion de terminos de prueba

El principal problema que ocurre al utilizar la función `consRec` definida anteriormente se da por el parámetro `notElem : Not (ElemLabel l ts)`. 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 retorna una computación 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 idéntica, 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"]

-- Las siguientes no compilan
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
```

3.2.4. 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:

```
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

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

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 _ 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)`. La implementación se describirá más adelante.

Su uso fue demostrado al comienzo de esta sección, con el siguiente caso:

```
persona : Record [('Clave', Nat), ('Nombre', String),
  ('Edad', Nat)]
persona = consRecAuto 'Clave' 3 $
  consRecAuto 'Nombre' 'Juan' $
  consRecAuto 'Edad' 27 $
  emptyRec
```

Con estas definiciones y técnicas es posible definir el record de arriba, sabiendo en tiempo de compilación que ninguna de las etiquetas utilizadas se repite.

En el caso de que las etiquetas si se repitan, mostrará un mensaje de error:

```
recordA : Record [('A', Nat), ('A', Nat)]
recordA = consRecAuto 'A' 10 $ consRecAuto 'A' 10 $ emptyRec

When checking right hand side of recordA with expected type
             Record [('A', Nat), ('A', Nat)]

     Type mismatch between
             RecordOrUnit [('A', Integer), ('A', Integer)]
               (Type of consRecAuto 'A' 10
                 (consRecAuto 'A' 10 emptyRec))
     and
             Record [('A', Nat), ('A', Nat)] (Expected type)
```

Como `RecordOrUnit` se computa a `()`, nunca puede unificarlo con `Record`.

3.3. Definición de un record

Un caso pendiente que quedó de la sección anterior es explicar la implementación misma de `consRecAuto`. Para hacerlo basta explicar la implementación del tipo `Record` utilizada en este trabajo:

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

Esta definición se corresponde a la de Haskell. Un record es una lista heterogénea donde sus etiquetas no tienen valores repetidos.

Con esta definición se puede ver cómo se implementa `emptyRec` y `consRecAuto`

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

Un record vacío simplemente tiene una lista heterogénea vacía y la prueba del caso base de etiquetas no repetidas

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

```
  mkTypeOrUnit (isLabelSet ((l, t) :: ts))
  (\isLabelSet => MkRecord isLabelSet (val :: hs))
```

La extensión de un record genera la prueba de que la lista resultante no tiene etiquetas repetidas, y crea el nuevo record con esa prueba y la lista heterogénea vieja con el nuevo elemento.

3.3.1. 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.

Se decidio utilizar la siguiente solución, en donde `HList` no solo tiene el tipo en su lista, sino la etiqueta también:

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

La implementación es idéntica a la de Idris, con la diferencia de que se debe pasar no solo el valor sino la etiqueta también.

Un ejemplo sería el siguiente:

```
[("Clave", 1), ("Nombre", "Juan")] :
  HList [("Clave", Nat), ("Nombre", String)]
```

3.4. Implementación de operaciones sobre records

Al comienzo de esta sección vimos algunas operaciones sobre records y su uso. Ahora describiremos mejor cómo se implementaron éstas.

3.4.1. Proyección sobre un record

El ejemplo visto anteriormente fue el siguiente:

```
personaYDireccion : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat), ("Direccion", String)]

proyeccion: Record [("Clave", Nat), ("Direccion", String)]
proyeccion = hProjectByLabelsAuto ["Clave", "Direccion"]
  personaYDireccion
```

La proyección sobre un record toma una lista de etiquetas, y retorna solo los campos del record asociados a esas etiquetas, es decir, realiza una proyección del record.

Esta función utiliza el mismo truco de `consRecAuto`. Para poder proyectar una lista de campos sobre un record, es necesario que esos campos no tengan etiquetas repetidas tampoco.

```
proyeccion: Record [("Clave", Nat), ("Clave", Nat)]
proyeccion = hProjectByLabelsAuto ["Clave", "Nat"]
-- No compila
```

El tipo de tal función es el siguiente:

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

Esta función toma una lista de etiquetas, un record, y automáticamente genera una prueba de `IsSet ls`. En caso de poder hacerlo, retorna un nuevo record `Record (proyectLeft ls ts)`.
`projectLeft` es una función a nivel de tipos. Permite retornar una lista a nivel de tipos distinta según los campos a proyectar.

```
projectLeft ["Clave"]
 [("Clave", Nat), ("Nombre", String)] =
 [("Clave", Nat)]
projectLeft ["Clave", "Direccion"]
 [("Clave", Nat), ("Nombre", String), ("Direccion", String)] =
 [("Clave", Nat), ("Direccion", String)]
```

Esta es una función común que se puede usar sobre listas, pero puede ser también utilizada para tener distintos tipos en un record

```
Record (projectLeft ["Clave", "Direccion"]
 [("Clave", Nat), ("Nombre", String), ("Direccion", String)]) =
Record [("Clave", Nat), ("Direccion", String)]
```

La computación simplemente se translada a los tipos.
La implementación de la función es la siguiente:

```
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

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 implementación hace recursión sobre los campos del record. Si el primer campo pertenece a la lista a proyectar, entonces lo retorna. Si no pertenece, no lo retorna. Si pertenece, elimina ese campo de la lista a proyectar y aplica el mismo razonamiento al resto de la lista.

Con esto se puede implementar `hProjectByLabelsAuto`

```
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)
```

Simplemente aplica el mismo truco de `consRecAuto`, delegando la llamada a `hProjectByLabels`

```
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
```

Para entender esta implementación iremos por partes:

```
isLabelSet = recLblIsSet rec
hs = recToHList rec
```

Dado un record, se obtiene su lista heterogénea y la prueba de que no tiene etiquetas repetidas, utilizando las siguientes funciones:

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

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

```
(lsRes ** (hsRes, prjLeftRes)) =
      fst $ hProjectByLabelsHList ls hs
```

Esta llamada hace uso de la siguiente función:

```
hProjectByLabelsHList : DecEq lty => {ts : LabelList lty} ->
  (ls : List lty) -> HList ts ->
  ((ls1 : LabelList lty ** (HList ls1, IsProjectLeft ls ts ls1)),
  (ls2 : LabelList lty ** (HList ls2, IsProjectRight ls ts ls2)))
```

Esta función realiza la proyección a nivel de HList. También contiene otro mecanismo del manejo de tipos: En vez de retornar la computación en el tipo mismo, retorna un predicado que representa esa computación.

El tipo `IsProjectLeft` es el tipo que cumple la siguiente propiedad:

```
IsProjectLeft ls ts1 ts2 <-> ts2 = projectLeft ls ts1
```

Es la representación de la proposición *'Si se proyecta ls sobre ts1, el resultado es ts2'*. `IsProjectRight` cumple el mismo propósito, pero realiza la proyección por la derecha (retorna todos los elementos que no fueron proyectados por `projectLeft`).

Su definición es la siguiente:

```
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)

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)
```

Como se ve, la definición de `IsProjectLeft` es muy similar a la de `projectLeft`. Esto es necesario para que es cumplan las propiedades descritas anteriormente: El predicado indica que su resultado corresponde a aplicar tal función.

Se pueden ver las similitudes comparando ambas implementaciones:

```
IPL_EmptyLabels : DecEq lty => IsProjectLeft {lty} [] ts []
projectLeft [] ts = []

IPL_EmptyVect : DecEq lty => IsProjectLeft {lty} ls [] []
projectLeft 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)
projectLeft ls ((l,ty) :: ts) | Yes lIsInLs =
    let delLFromLs = deleteElem ls lIsInLs
        rest = projectLeft delLFromLs ts
    in (l,ty) :: rest

IPL_ProjLabelNotElem : DecEq lty => Not (Elem l ls) ->
    IsProjectLeft {lty} ls ts res1 ->
    IsProjectLeft ls ((l,ty) :: ts) res1
projectLeft ls ((l,ty) :: ts) | No _ = projectLeft ls ts
```

Cada constructor del predicado se corresponde a cada caso de pattern matching de la definición de la función (sea pattern matching regular o utilizando `with`).

Algunos ejemplos son los siguientes:

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

Con estas definiciones, los siguientes dos son equivalentes:

```
(ls1 : LabelList lty ** (HList ls1, IsProjectLeft ls ts ls1)

HList (projectLeft ls ts)
```

En este trabajo se decidió utilizar los predicados para operaciones internas porque resulta más sencillo realizar pattern matching.

Cada vez que se tiene un llamado a una función que retorna `projectLeft`, es necesario hacer pattern matching sobre los argumentos específicos que la implementación de esa función hace, en el orden exacto. Sin embargo, con predicados como `IsProjectLeft` solo basta hacer pattern matching sobre los constructores y nada más. El desarrollo queda más simple.

El resto de la implementación de `hProjectByLabelsHList` se puede ver en el apéndice.

Siguiendo con la implementación de `hProjectLabels`, se tiene lo siguiente:

```
isLabelSetRes =
  hProjectByLabelsLeftIsSet_Lemma2 prjLeftRes isLabelSet
```

Este lema es el siguiente:

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

Indica que si se hace proyección sobre una lista de campos, y si no hay etiquetas repetidas en los campos originales entonces tampoco lo van a haber en los resultantes. Su implementación se puede encontrar en el apéndice.

```
resIsProjComp = fromIsProjectLeftToComp prjLeftRes lsIsSet
```

La función es la siguiente:

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

Esta función representa la propiedad que debe cumplir `IsProjectLeft` en relacion a `projectLeft` que vimos anteriormente. Se le agrega el hecho de que `ls` no debe tener repetidos, que es necesario para que esta función pueda ser implementada. Su implementación también está en el apéndice.

```
recRes = hListToRec {prf=isLabelSetRes} hsRes
```

La función `hListToRec` construye un record a partir de una lista heterogénea y una prueba de etiquetas no repetidas. Su implementación es trivial:

```
hListToRec : DecEq lty => {ts : LabelList lty} ->
  {prf : IsLabelSet ts} -> HList ts -> Record ts
hListToRec {prf} hs = MkRecord prf hs
```

Por último se tiene esta línea de código:

```
in rewrite (sym resIsProjComp) in recRes
```

En Idris este es un caso de reescribir un término teniendo una prueba de igualdad. Por la llamada a `hProjectByLabelsHList` y `hListToRec` se tiene un término `recRes : Record ts2`. Por la llamada a `fromIsProjectLeftToComp` se tiene un término `resIsProjComp : ts2 = projectLeft ls ts1`. Por lo tanto, con esta técnica de reescritura se puede obtener un término de tipo `Record (projectLeft ls ts1)`, tal como lo indica el tipo de la función.

3.4.2. Búsqueda de un elemento en un record

Al comienzo de la sección vimos un ejemplo de buscar el valor de un campo en particular de un record

```
persona : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat)]

nombre : String
nombre = hLookupByLabelAuto "Nombre" persona
```

La funcionalidad de lookup se puede definir de forma muy similar a las anteriores. Primero se comienza con un predicado que indica que una lista de etiquetas contiene a otra etiqueta con un determinado tipo en particular

```
data HasField : (l : lty) -> LabelList lty ->
  Type -> Type where
  HasFieldHere : HasField l ((l,ty) :: ts) ty
  HasFieldThere : HasField l1 ts ty1 ->
    HasField l1 ((l2,ty2) :: ts) ty1
```

`HasField l ts ty` indica que en la lista de etiquetas `ts` existe la etiqueta `l` que tiene asociado el tipo `ty`. Un ejemplo de tal tipo es `HasField 'Edad' [('Nombre', String), ('Edad', Nat)] Nat`. Su definicion es recursiva, donde `HasFieldHere` indica que la etiqueta se encuentra en la cabeza de la lista, y `HasFieldThere` indica que se encuentra en la cola de la lista.

Una primera definicion es la de obtener el valor de una lista heterogenea:

```
hLookupByLabel_HList : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> HList ts -> HasField l ts ty -> ty
hLookupByLabel_HList _ (val :: _) HasFieldHere = val
hLookupByLabel_HList l (_ :: ts)
  (HasFieldThere hasFieldThere) =
  hLookupByLabel_HList l ts hasFieldThere
```

Esta funcion toma una etiqueta `l : lty`, una lista heterogenea `HList ts`, y una prueba de que esa etiqueta pertenece a esa lista con `HasField l ts ty`. Con tales dato permite retornar un valor del tipo `ty`, el cual es el asociado a tal etiqueta.

Su implementacion realiza pattern matching sobre el predicado, obteniendo el valor `val : ty` si la etiqueta esta en la cabeza de la lista o realizando un llamado recursivo sino.

La funcion que obtiene un elemento de un record es sencilla

```
hLookupByLabel : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> Record ts -> HasField l ts ty -> ty
hLookupByLabel {ts} {ty} l rec hasField =
  hLookupByLabel_HList {ts} {ty} l (recToHList rec) hasField
```

Como la definicion de record de este trabajo contiene la lista heterogenea dentro de el, entonces esta funcion simplemente aplica la funcion definida anteriormente a tal lista heterogenea del record.

Al igual que las funcionalidades anteriores, se puede definir una funcion que calcule el predicado de forma automatica en tiempo de compilacion. Sin embargo, con el caso de lookup ocurre un problema. Idealmente, uno quisiera tener la siguiente funcion:

```
hasField : Deceq lty => (l : lty) ->
  (ts : LabelList lty) -> (ty : Type) ->
  Dec (HasField l ts ty)
```

Con esta funcion uno podria definir la funcion de calculo automatico de la siguiente forma

```
hLookupByLabelAuto : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> Record ts ->
  TypeOrUnit (hasField l ts ty) ty
hLookupByLabelAuto {ts} {ty} l rec =
  mkTypeOrUnit (hasField l ts ty)
  (\tsHasL => hLookupByLabel {ts} {ty} l rec tsHasL)
```

El problema ocurre que no es posible definir `hasField`.

Para implementar `hasField`, se tiene una variable `ty : Type` que puede ser cualquier tipo. A su vez, si la etiqueta `l : lty` pertenece a la lista, entonces en la lista la etiqueta va a tener asociado un tipo `ty2 : Type`, que puede ser cualquier otro. El problema ocurre en que no se puede verificar la igualdad de los tipos `ty` y `ty2`, ya que no existe forma de igualar valores de tipo `Type`.

Por ejemplo, se puede tener `hasField 'Edad' [('Edad', Nat)] Nat` o `hasField 'Edad' [('Edad', Nat)] String`. El primer caso deberia retornar `Yes` con una prueba, mientras que el segundo deberia retornar `No` con una prueba de su opuesto, ya que el tipo `Nat` y `String` son distintos, por lo que el predicado falla. Sin embargo, no es posible realizar un chequeo `Nat = Nat` o `Not (Nat = String)`, ya que eso requeriria que existiese una instancia de `DecEq Type`, la cual no existe y no es posible definir.

El hecho de que no existe una instancia `DecEq Type` en el lenguaje tiene varias razones:

- Dados dos tipos cualquiera en un lenguaje de tipos dependientes, verificar si son iguales no es decidible. Es decir, teoricamente, es imposible crear una instancia de `DecEq Type`
- Si existiera, no existiria el polimorfismo parametrico. El polimorfismo parametrico permite abstraerse de un tipo y trabajar con el sin saber especificamente cual es. Esto permite a uno deducir propiedades de la funcion solamente conociendo el tipo. Por ejemplo, si se tiene la funcion `id : a ->a`, se puede deducir que la unica posible implementacion es `id val = val`, ya que es imposible que la funcion pueda conocer informacion sobre el tipo `a : Type`.

  Sin embargo, si existiera una instancia de `DecEq Type`, entonces la siguiente funcion seria valida

```
id2 : a -> a
id2 {a=a} val with (decEq a Nat)
  id2 {a=Nat} val | Yes aIsNat = 10
  id2 {a=a} val | No notAIsNat = val
```

La funcion `id2 : a ->a` retorna el mismo valor para todos los tipos, menos para el tipo `Nat`, donde retorna siempre el valor 10. Esta funcion contradice las garantias de parametricidad del tipo `a ->a`.

A pesar de no poder utilizar `TypeOrUnit`, es posible definir una funcion que calcule el predicado de forma automatica utilizando la funcionalidad `auto` de Idris, de esta forma

```
hLookupByLabelAuto : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> Record ts ->
  {auto hasField : HasField l ts ty} -> ty
hLookupByLabelAuto {ts} {ty} l rec {hasField} =
  hLookupByLabel_HList {ts} {ty} l (recToHList rec) hasField
```

`auto hasField : HasField l ts ty` indica que el typechecker de Idris va a intentar generar el predicado `HasField l ts ty` de forma automatica. Idris se da cuenta de que la definicion de `HasField` es inductiva y tiene distintos constructores, por lo que intenta, mediante fuerza bruta, aplicar los constructores secuencialmente hasta encontrar un valor que tenga el tipo deseado.

Por ejemplo, si se quisiera obtener una prueba de `HasField 'Edad' [('Nombre', String), ('Edad', Nat)] Nat`, Idris primero va a construir `HasFieldHere`. Como `HasFieldHere` no puede unificarse con el tipo deseado, sigue con el siguiente caso. Proximamente, Idris intenta construir `HasFieldThere HasFieldHere`. Idris consigue unificar el tipo de ese valor con el esperado, entonces proporciona el valor `HasFieldThere HasFieldHere` a la funcion `hLookupByLabelAuto`.

Como conclusion, utilizar `TypeOrUnit` es muy util para definir estas funciones con calculo automatico, pero solo puede usarse si el predicado no depende de un valor `a : Type`. Si es asi, se deben utilizar funcionalidades propias del lenguaje como `auto`.

3.4.3. Unión izquierda

En esta sección describiremos la unión por izquierda de records.

Al comienzo de la sección se vió el siguiente ejemplo:

```
persona : Record [("Clave", Nat), ("Nombre", String),
  ("Edad", Nat)]
persona = hLeftUnion personaConNombre personaConEdad
```

`hLeftUnion` toma dos records cualesquiera, y retorna uno nuevo con todos los campos del de la izquierda, más los campos del de la derecha que no están repetidos en el de la izquierda. Básicamente realiza una unión de los campos de ambos records, pero si hay campos con etiquetas repetidas toma el valor del de la izquierda.

Su definición es la siguiente:

```
hLeftUnion : DecEq lty => {ts1, ts2 : LabelList lty} ->
  Record ts1 -> Record ts2 -> Record (hLeftUnion_List ts1 ts2)
hLeftUnion ts1 ts2 =
  let (tsRes ** (resUnion, isLeftUnion)) =
        hLeftUnionPred ts1 ts2
      leftUnionEq = fromHLeftUnionPredToFunc isLeftUnion
  in rewrite (sym leftUnionEq) in resUnion
```

Antes que nada, al obtener un record nuevo se debe computar la lista de sus campos. Esto se hace con la función `hLeftUnion_List`:

```
deleteLabelAt : DecEq lty => lty -> LabelList lty ->
  LabelList lty
deleteLabelAt l [] = []
deleteLabelAt l1 ((l2,ty) :: ts) with (decEq l1 l2)
  deleteLabelAt l1 ((l2,ty) :: ts) | Yes l1EqL2 = ts
  deleteLabelAt l1 ((l2,ty) :: ts) | No notL1EqL2 =
    (l2,ty) :: deleteLabelAt l1 ts

deleteLabels : DecEq lty => List lty -> LabelList lty ->
  LabelList lty
deleteLabels [] ts = ts
deleteLabels (l :: ls) ts =
  let subDelLabels = deleteLabels ls ts
  in deleteLabelAt l subDelLabels

hLeftUnion_List : DecEq lty => LabelList lty ->
  LabelList lty -> LabelList lty
hLeftUnion_List ts1 ts2 =
  ts1 ++ (deleteLabels (labelsOf ts1) ts2)
```

Esta función es un poco más compleja que las anteriores. Esta función define a la unión de dos listas tomando la primera, y agregandole todos los campos de la segunda, pero primero eliminandole los campos de la primera.

Esta eliminación se realiza con `deleteLabels`, que simplemente recorre toda la lista de campos a eliminar y lo elimina (si pertenece a la segunda lista).

Al igual que en `hProjectByLabels`, `hLeftUnion` hace uso de un predicado que representa la computación de la unión de los campos del record, y realiza la unión misma utilizando ese predicado.

```
hLeftUnionPred : DecEq lty => {ts1, ts2 : LabelList lty} ->
  Record ts1 -> Record ts2 ->
  (tsRes : LabelList lty ** (Record tsRes,
    IsLeftUnion ts1 ts2 tsRes))
```

Al igual que `IsProjectLeft`, `IsLeftUnion` es el predicado que cumple estas propiedades:

```
IsLeftUnion ts1 ts2 ts3 -> ts3 = hLeftUnion_List ts1 ts2
```

```
IsLeftUnion ts1 ts2 (hLeftUnion_List ts1 ts2)
```

Una de esas propiedades es la utilizada en la siguiente función vista más arriba:

```
fromHLeftUnionFuncToPred : DecEq lty =>
  {ts1, ts2 : LabelList lty} ->
  IsLeftUnion ts1 ts2 (hLeftUnion_List ts1 ts2)
```

A continuación se muestra la definición del tipo `IsLeftUnion`, que al igual que `IsProjectLeft`, se corresponde uno a uno con la definición de su función análoga (`hLeftUnion_List` en este caso):

```
data DeleteLabelAtPred : DecEq lty => lty -> LabelList lty ->
  LabelList lty -> Type where
  EmptyRecord : DecEq lty => {l : lty} ->
    DeleteLabelAtPred l [] []
  IsElem : DecEq lty => {l : lty} ->
    DeleteLabelAtPred l ((l,ty) :: ts) ts
  IsNotElem : DecEq lty => {l1 : lty} -> Not (l1 = l2) ->
    DeleteLabelAtPred l1 ts1 ts2 ->
    DeleteLabelAtPred l1 ((l2,ty) :: ts1) ((l2,ty) :: ts2)

data DeleteLabelsPred : DecEq lty => List lty -> LabelList lty ->
  LabelList lty -> Type where
  EmptyLabelList : DecEq lty =>
    DeleteLabelsPred {lty=lty} [] ts ts
  DeleteFirstOfLabelList : DecEq lty =>
    DeleteLabelAtPred l tsAux tsRes ->
    DeleteLabelsPred ls ts tsAux ->
    DeleteLabelsPred {lty=lty} (l :: ls) ts tsRes

data IsLeftUnion : DecEq lty => LabelList lty -> LabelList lty ->
  LabelList lty -> Type where
  IsLeftUnionAppend : DecEq lty =>
    {ts1, ts2, ts3 : LabelList lty} ->
    DeleteLabelsPred (labelsOf ts1) ts2 ts3 ->
    IsLeftUnion ts1 ts2 (ts1 ++ ts3)
```

El resto de la implementación de `hLeftUnion` se puede ver en el apéndice.

3.5. Comparación con otros lenguajes

En esta sección compararemos esta implementación de records extensibles en Idris con las vistas en la sección de *'Estado del Arte'* para otros lenguajes.

3.5.1. Elm

El ejemplo visto para Elm era 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
  }
```

Este código puede ser escrito en Idris de la siguiente forma:

```
Positioned : LabelList String -> LabelList String
Positioned ts = ("x", Double) :: ("y", Double) :: ts

Named : LabelList String -> LabelList String
Named ts = ("name", String) :: ts

Moving : LabelList String -> LabelList String
Moving ts = ("velocity", Double) :: ("angle", Double) :: ts

lady : Record (Named [("age", Nat)])
lady = consRecAuto "name" "Lois Lane" $
  consRecAuto "age" 31 $
  emptyRec

dude : Record (Named . Moving . Positioned $ [])
dude = consRecAuto "name" "Clark Kent" $
  consRecAuto "velocity" 42 $
  consRecAuto "angle" 30 $
  consRecAuto "x" 0 $
  consRecAuto "y" 0 $
  emptyRec
```

Esta es la traducción más directa del ejemplo de Elm. Sin embargo, si se quiere mejorar el diseño, se puede utilizar la flexibilidad de Idris para diseñarlo de otra forma (utilizando typeclasses, tipos de datos, etc).

3.5.2. Purescript

El ejemplo visto para Purescript era el siguiente:

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

Esto puede traducirse a Idris de la siguiente forma:

```
fullName : {ts : LabelList String} ->
  {auto hasFirst : HasField "firstName" ts String} ->
  {auto hasLast : HasField "lastName" ts String} ->
  Record ts -> String
fullName {hasFirst} {hasLast} rec =
  let firstName = hLookupByLabel "firstName" rec hasFirst
      lastName = hLookupByLabel "lastName" rec hasLast
  in firstName ++ " " ++ lastName
```

Un ejemplo de su uso sería el siguiente:

```
persona : Record [("firstName", String), ("age", Nat),
  ("lastName", String)]
persona = consRecAuto "firstName" "Juan" $
  consRecAuto "age" 23 $
  consRecAuto "lastName" "Sanchez" $
  emptyRec

fullName persona : String
-- "Juan Sanchez"
```

En Idris se puede parametrizar por cualquier predicado, en particular por `HasField`, permitiendo la programación con *row polymorphism* de forma similar a Purescript y otros lenguajes.

Comparando con Elm y Purescript, al ser todos los resultados del tipo `Record` se pueden utilizar todas las operaciones vistas hasta ahora en esta sección, lo cual estos lenguajes no soportan.

Capítulo 4

Caso de estudio

En el capítulo anterior se describió el diseño e implementación de records extensibles en Idris realizado en este trabajo. Sin embargo, el único uso de tales records visto fue en algunos ejemplos básicos. En el siguiente capítulo se mostrará un caso de estudio en el cual se utilizaron los records extensibles de este trabajo para poder solucionarlo.

4.1. Descripción del caso de estudio

El caso de estudio consiste en un pequeño lenguaje aritmético con variables y constantes. Se espera poder definir este lenguaje en Idris como un DSL (*Domain Specific Language*), poder crear términos de este lenguaje, y luego evaluarlos en Idris y obtener valores naturales que se correspondan a la expresión aritmética definida.

Este lenguaje *Exp* se define de la siguiente manera:

- Si *n* es un natural, entonces *n* pertenece a *Exp* como un literal.
- Si *x* es un string, entonces *x* pertenece a *Exp* como una variable.
- Si *e1* y *e2* pertenecen a *Exp*, entonces *e1 + e2* pertenece a *Exp* como una suma.
- Si *x* es un string, *n* es un natural y *e* pertenece a *Exp*, entonces *let x := n in e* pertenece a *Exp* como una sustitución de una variable.

Este lenguaje permite definir expresiones aritméticas donde se pueden definir valores numéricos, variables, o sumar expresiones. Los siguientes son ejemplos de expresiones de este lenguaje:

```
x
x + 3
let x := 3 in x + 3
let y := 10 in (let x := 3 in x + 3)
```

Su gramática en BNF es:
PENDIENTE

Se decidió utilizar records extensibles e Idris en este caso de estudio para poder verificar que las expresiones estén bien formadas, y que las llamadas al evaluador sean válidas, en tiempo de compilación.

Si se tiene una expresión, como `x + 3`, para evaluar esa expresión es necesario tener un ambiente con valores para las variables libres. En este caso es necesario tener un ambiente con el valor de la variable `x`, de lo contrario la evaluación de esa expresión no es válida. Con Idris, es posible verificar en tiempo de compilación que cualquier ambiente utilizado para evaluar una expresión debe contener valores para las variables libres de ésta. Con esta implementación, al llamar al evaluador se verifican que los tipos sean correctos.

Los records extensibles fueron utilizados para poder definir tal ambiente al momento de evaluar una expresión. Como el ambiente contiene una lista de variables con sus valores, éstos pueden ser representados como un record, donde cada etiqueta es una variable libre, y éste puede ser extendido con nuevas variables, o se le pueden eliminar variables fuera de alcance. Al ser un record extensibles se tiene la garantía de que ninguna variable se encuentra repetida en el ambiente, y en cada momento de la implementación se tiene una prueba de que toda variable libre pertenece al ambiente.

Sin records extensibles ni tipos dependientes, tales garantías no existen en tiempo de compilación. Al momento de llamar al evaluador con una expresión en particular, el compilador no puede saber si las variables libres pertenecen al ambiente utilizado o no. Si alguna no pertenece, la evaluación va a fallar en tiempo de ejecución. Sin records extensibles tampoco se tiene una estructura de datos adecuada (con las propiedades deseadas) para almacenar las variables en el ambiente.

4.2. Definición de una expresión

Las expresiones de este lenguaje deben seguir las siguientes reglas de buena formación:

PENDIENTE

A su vez, toda expresión tiene un conjunto de variables libres, que se definen con las siguientes reglas:

PENDIENTE

Estas reglas pueden implementarse en Idris en un único tipo `Exp`:

```
data VarDec : String -> Type where
  (:=) : (var : String) -> Nat -> VarDec var

data Exp : List String -> Type where
  Add : Exp fvs1 -> Exp fvs2 ->
    IsLeftUnion_List fvs1 fvs2 fvsRes ->
    Exp fvsRes
  Var : (l : String) -> Exp [l]
  Lit : Nat -> Exp []
  Let : VarDec var -> Exp fvsInner ->
    DeleteLabelAtPred_List var fvsInner fvsOuter ->
    Exp fvsOuter
```

Un valor de tipo `VarDec x` contiene la variable `x` y un natural. Representa la declaración de una variable, como `'x' := 10 : VarDec 'x'`.

El tipo `Exp` se define como un tipo parametrizado por una lista de variables. Esta lista representa la lista de variables libres que ocurren en la expresión. Un valor de tipo `Exp ['x', 'y']` es una expresión que tiene a *'x'* e *'y'* como variables libres. Se decidió incluir la información de variables libres en el tipo para poder facilitar tanto la implementación del evaluador, como la construcción de expresiones.

Cada constructor representa cada regla de formación de una expresión. A su vez, cada constructor sigue las reglas de construcción de variables libres definidas anteriormente

```
Var : (l : String) -> Exp [l]
```

Crea una expresión con una variable. En el tipo retorna esa variable como libre.

```
Lit : Nat -> Exp []
```

Crea una expresión con un número como literal. No retorna variables libres.

```
Add : Exp fvs1 -> Exp fvs2 ->
    IsLeftUnion_List fvs1 fvs2 fvsRes ->
    Exp fvsRes
```

Toma dos subexpresiones y crea una nueva expresión que representa la suma. Las variables libres son la unión por izquierda de la primera subexpresión con las de la segunda subexpresión.

`IsLeftUnion_List` es un predicado idéntico a `IsLeftUnion` visto en el capítulo anterior, solo que este funciona sobre listas `List lty` en vez de listas de campos con etiquetas `LabelList lty`.

```
data DeleteLabelAtPred_List : lty -> List lty ->
  List lty -> Type where
  EmptyRecord_List : {l : lty} -> DeleteLabelAtPred_List l [] []
  IsElem_List : {l : lty} -> DeleteLabelAtPred_List l (l :: ls) ls
  IsNotElem_List : {l1 : lty} -> Not (l1 = l2) ->
    DeleteLabelAtPred_List l1 ls1 ls2 ->
    DeleteLabelAtPred_List l1 (l2 :: ls1) (l2 :: ls2)

data DeleteLabelsPred_List : List lty -> List lty ->
  List lty -> Type where
  EmptyLabelList_List : DeleteLabelsPred_List {lty} [] ls ls
  DeleteFirstOfLabelList_List : DeleteLabelAtPred_List l lsAux lsRes ->
    DeleteLabelsPred_List ls1 ls2 lsAux ->
    DeleteLabelsPred_List {lty} (l :: ls1) ls2 lsRes

data IsLeftUnion_List : List lty -> List lty ->
  List lty -> Type where
  IsLeftUnionAppend_List :
    {ls1, ls2, ls3 : List lty} ->
```

```
    DeleteLabelsPred_List ls1 ls2 ls3 ->
    IsLeftUnion_List ls1 ls2 (ls1 ++ ls3)
```

Un ejemplo sería el siguiente:

```
IsLeftUnion_List ["A", "B"] ["B", "C"]
  ["A", "B", "C"]
```

```
Let : VarDec var -> Exp fvsInner ->
    DeleteLabelAtPred_List var fvsInner fvsOuter ->
    Exp fvsOuter
```

Crea una expresión tomando una subexpresión, y una asignación de un valor a una variable. Las variables libres de esta expresión se corresponden a las variables libres de la subexpresión, eliminando la variable recientemente asignada.

`DeleteLabelAtPred_List` es el predicado que indica esto. Se comporta igual que `DeleteLabelAtPred` definido en la sección anterior, solo que se aplica sobre listas `List lty` en vez de listas de etiquetas `LabelList lty`. Su definición se mostró más arriba.

Un ejemplo sería el siguiente:

```
DeleteLabelAtPred_List "A" ["A", "B", "C"]
  ["B", "C"]
```

Al igual que en el capítulo anterior, se decidió utilizar predicados que representan las computaciones de unión por izquierda y eliminación de una etiqueta porque simplifican el desarrollo, permitiendo realizar pattern matching de forma más sencilla.

4.2.1. Construcción de una expresión

Para poder construir expresiones de este lenguaje, se definieron funciones auxiliares que construyen valores del tipo `Exp`.

```
var : (l : String) -> Exp [l]
var l = Var l

lit : Nat -> Exp []
lit n = Lit n

add : Exp fvs1 -> Exp fvs2 -> Exp (leftUnion fvs1 fvs2)
add {fvs1} {fvs2} e1 e2 = Add e1 e2
  (fromLeftUnionFuncToPred {ls1=fvs1} {ls2=fvs2})

eLet : VarDec var -> Exp fvs -> Exp (deleteAtList var fvs)
eLet {var} {fvs} varDec e = Let varDec e
  (fromDeleteLabelAtListFuncToPred {l=var} {ls=fvs})
```

Las cuatro funciones simplemente realizan la aplicación del constructor correspondiente. `add` y `eLet` tienen la particularidad de que realizan las computaciones

sobre las listas de variables, y construyen los predicados correspondientes dada esas computaciones. `leftUnion` y `deleteAtList` son las computaciones análogas a `IsLeftUnion_List` y `DeleteAtLabel_List`, cumpliendo con las siguientes propiedades:

```
ls2 = deleteAtList l ls1 <-> DeleteLabelAtPred_List l ls ls2

ls3 = leftUnion ls1 ls2 <-> IsLeftUnion_List ls1 ls2 ls3
```

Las funciones `fromLeftUnionFuncToPred` y `fromDeleteLabelAtListFuncToPred` son las que representan las propiedades anteriores:

```
fromDeleteLabelAtListFuncToPred : DecEq lty => {l : lty} ->
  {ls : List lty} -> DeleteLabelAtPred_List l ls (deleteAtList l ls)

fromLeftUnionFuncToPred : DecEq lty => {ls1, ls2 : List lty} ->
  IsLeftUnion_List {lty} ls1 ls2 (leftUnion ls1 ls2)
```

A continuación se muestran ejemplos de expresiones construidas con las funciones anteriores:

```
exp1 : Exp ["x", "y"]
exp1 = add (var "x") (add (lit 1) (var "y"))

exp2 : Exp ["y"]
exp2 = eLet ("x" := 10) $ add (var "x") (var "y")

exp3 : Exp []
exp3 = eLet ("y" := 5) exp4
```

También se decidió incluir otra función útil para construir expresiones. Se trata de *local*, que permite definir un binding local de declaración de variables. Su uso sería el siguiente:

```
exp1 : Exp []
exp1 = local ["x" := 10] $ cons 1

exp2 : Exp []
exp2 = local ["x" := 10, "y" := 9] $ add (var "x") (var "y")
```

`local` permite declarar varias variables simultáneamente.
Su implementación es la siguiente:

```
data LocalVariables : List String -> Type where
  Nil : LocalVariables []
  (::) : VarDec l -> LocalVariables ls ->
    LocalVariables (l :: ls)

localPred : (vars : LocalVariables localVars) ->
```

```
    (innerExp : Exp fvsInner) -> {isSet : IsSet localVars} ->
    Exp (deleteList localVars fvsInner)

local : (vars : LocalVariables localVars) -> (innerExp : Exp fvsInner) ->
  TypeOrUnit (isSet localVars) (Exp (deleteList localVars fvsInner))
local {localVars} {fvsInner} vars innerExp =
  mkTypeOrUnit (isSet localVars)
    (\localIsSet => localPred vars innerExp {isSet=localIsSet})
```

Antes que nada, `local` utiliza el mismo truco de `consRecAuto` y otras funciones sobre records con `TypeOrUnit`. En este caso, `local` permite construir una prueba de `IsSet localVars` automáticamente en tiempo de compilación. Esta prueba evita que se defina la misma variables varias veces, como `local ['x' := 1, 'x' := 2]`.

Tal declaración de variables se define en `LocalVariables`, que es un tipo que contiene la lista de declaraciones locales de variables, y en su tipo mantiene la lista de variables declaradas. Por ejemplo, se tiene `['x' := 10, 'y' := 4] : LocalVariables ['x', 'y']`.

La expresión resultante calcula sus variables libres con `deleteList`. Esta función es la análoga a `DeleteLabelsPred_List` que cumple esta propiedad:

```
ls3 = deleteList ls1 ls2 <-> DeleteLabelsPred_List ls1 ls2 ls3
```

La implementación de `localPred` se puede ver en el anexo, pero básicamente realiza una construcción secuencial de `Let`s para cada variable declarada en el binding. Básicamente, las siguientes expresiones son equivalentes:

```
local ["x" := 10] $ cons 1 <-> eLet ("x" := 10) $ cons 1

local ["x" := 10, "y" := 9] $ add (var "x") (var "y") <->
 eLet ("x" := 10) $ eLet ("y" := 9) $ add (var "x") (var "y")
```

4.3. Evaluación de una expresión

La evaluación de una expresión se realiza con esta función:

```
interpEnv : Ambiente fvsEnv -> IsSubSet fvs fvsEnv -> Exp fvs -> Nat
```

Dada una expresión `Exp fvs`, con una lista de variables libres `fvs`, se necesita tener un ambiente `Ambiente fvsEnv` con variables `fvsEnv`. El ambiente contiene los valores para todas esas variables, las cuales deben incluir a todas las variables libres de la expresión.

Si se tiene `Exp ['x', 'y']`, entonces se puede tener `Ambiente ['x', 'y']`, o `Ambiente ['x', 'y', 'z']`, o `Ambiente ['x', 'w', 'y', 'z']`. El ambiente puede tener variables extras, pero siempre debe tener valores para las variables libres de la expresión. Esto garantiza que cualquier llamada a `interpEnv` sea válida, ya que toda expresión puede evaluarse si se tienen valores para sus variables libres.

El tipo `IsSubSet fvs fvsEnv` refleja esa relación, como se puede ver en estos ejemplos:

```
IsSubSet ['x'] ['x']
IsSubSet ['x'] ['x', 'y']
IsSubSet ['x', 'y'] ['x', 'y', 'z']
```

Su definición es la siguiente:

```
data IsSubSet : List lty -> List lty -> Type where
  IsSubSetNil : IsSubSet [] ls
  IsSubSetCons : IsSubSet ls1 ls2 -> Elem l ls2 ->
    IsSubSet (l :: ls1) ls2
```

En el caso base la lista vacía es subconjunto de cualquier posible lista. En el caso recursivo, si un elemento a agregar pertenece a la lista, entonces agregando ese elemento a un subconjunto de esa lista también es un subconjunto.

Por último se define el ambiente de esta forma:

```
AllNats : List lty -> LabelList lty
AllNats [] = []
AllNats (x :: xs) = (x, Nat) :: AllNats xs

data Ambiente : List String -> Type where
  MkAmbiente : Record {lty=String} (AllNats ls) -> Ambiente ls
```

`AllNats` es una función que toma una lista de etiquetas y les asigna el tipo `Nat` a todas. Por ejemplo:

```
AllNats ['x', 'y'] = [('x', Nat), ('y', Nat)]
```

Esto permite poder utilizar el record `Record {lty=String} (AllNats ls)`. El ambiente se trata de un record extensible, donde sus etiquetas son strings, y sus campos siempre son de tipo `Nat`.

Un ejemplo (con una pseudo-sintaxis) sería:

```
{ 'x': 10, 'y': 20, 'z': 22 } :
  Record [('x', Nat), ('y', Nat), ('z', Nat)]
```

Esta definición del evaluador también permite definir la evaluación de expresiones cerradas (sin variables libres) de forma sencilla:

```
interp : Exp [] -> Nat
interp = interpEnv (MkAmbiente {ls=[]} emptyRec) IsSubSetNil
```

Ejemplos de tal evaluación serían los siguientes:

```
interp $ local ["x" := 10, "y" := 9] $ add (var "x") (var "y")
-- 19
```

```
interp $ eLet ("x" := 10) $ add (var "x") (lit 2)
-- 12

interp $ add (lit 1) (lit 2)
-- 3
```

La implementación del evaluador mismo se muestra a continuación:

```
interpEnv : Ambiente fvsEnv -> IsSubSet fvs fvsEnv -> Exp fvs -> Nat
interpEnv env subSet (Add e1 e2 isUnionFvs) =
  let (subSet1, subSet2) =
        ifIsSubSetThenLeftUnionIsSubSet subSet isUnionFvs
      res1 = interpEnv env subSet1 e1
      res2 = interpEnv env subSet2 e2
  in res1 + res2
interpEnv {fvsEnv} (MkAmbiente rec) subSet (Var l) =
  let hasField = HasFieldHere {l} {ty = Nat} {ts = []}
      hasFieldInEnv = ifIsSubSetThenHasFieldInIt  subSet hasField
  in hLookupByLabel l rec hasFieldInEnv
interpEnv env subSet (Cons c) = c
interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    with (isElem var fvsEnv)
  interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | Yes varInEnv =
    let
        (MkAmbiente recEnv) = env
        hasField = ifIsElemThenHasFieldNat varInEnv
        newRec = hUpdateAtLabel var n recEnv hasField
        newEnv = MkAmbiente newRec

        consSubSet =
          ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
        newSubSet = ifConsIsElemThenIsSubSet consSubSet varInEnv
    in interpEnv newEnv newSubSet e

  interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | No notVarInEnv =
    let (MkAmbiente recEnv) = env
        newRec = consRec var n recEnv
          {notElem = ifNotElemThenNotInNats notVarInEnv}
        newEnv = MkAmbiente newRec {ls = (var :: fvsEnv)}
        newSubSet =
          ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
    in interpEnv newEnv newSubSet e
```

Se explicará su implementación para cada tipo de expresión por separado.

4.3.1. Expresiones de literales

El caso de una expresión literal es el mas sencillo:

```
interpEnv env subSet (Lit c) = c
```

Evaluar una expresión literal significa retornar tal valor literal como un natural.

4.3.2. Expresiones con declaración de variable

Una expresión que declara una variable, para poder ser evaluada, necesita tener esa variable en el ambiente y obtener el valor de ella.

```
interpEnv {fvsEnv} (MkAmbiente rec) subSet (Var l) =
  let hasField = HasFieldHere {l} {ty = Nat} {ts = []}
      hasFieldInEnv = ifIsSubSetThenHasFieldInIt subSet hasField
  in hLookupByLabel l rec hasFieldInEnv
```

Como el ambiente es un record `Record (AllNats fvsEnv)`, es necesario obtener la prueba de que `l` pertenece a ese record, y luego hacer un lookup.

Primero, se obtiene una prueba de `HasField l [(l, Nat)] Nat` realizando esta llamada:

```
hasField = HasFieldHere {l} {ty = Nat} {ts = []}
```

Luego se hace uso de la siguiente llamada para obtener la prueba de que `l` pertenece al ambiente:

```
ifIsSubSetThenHasFieldInIt : DecEq lty => {ls1, ls2 : List lty} ->
  IsSubSet ls1 ls2 -> HasField l (AllNats ls1) Nat ->
  HasField l (AllNats ls2) Nat
```

Esta función indica que si un elemento pertenece a una lista `ls1` y tiene tipo `Nat`, y esa lista `ls1` es un subconjunto de otra lista `ls2`, entonces ese elemento también pertenece a la lista `ls2` con tipo `Nat`. Básicamente es un teorema que prueba que la propiedad de pertenencia a una lista (de tipos `Nat`) se preserva.

Esta se realiza en la siguiente llamada:

```
hasFieldInEnv = ifIsSubSetThenHasFieldInIt subSet hasField
```

`hasFieldInEnv` tiene tipo `HasField l (AllNats fvsEnv) Nat`. Como el record del ambiente tiene tipo `Record (AllNats fvsEnv)`, entonces se tiene una prueba de que el elemento pertenece al ambiente, por lo que se puede retornar su valor con la siguiente invocación:

```
hLookupByLabel l rec hasFieldInEnv
```

En este caso la llamada a lookup es válida en tiempo de compilación. Por cómo se definió el evaluador, toda variable libre de la expresión pertenece al ambiente, por lo cual es posible probar que una variable específica de `Var l` pertenece al ambiente, y por lo tanto obtener su valor. En ningún momento el lookup va a fallar en tiempo de ejecución porque no pudo encontrar la variable.

4.3.3. Expresiones con suma

Una expresión del tipo suma es sencilla de evaluar. Se necesitan evaluar las subexpresiones de ella, y luego sumar sus resultados:

```
interpEnv env subSet (Add e1 e2 isUnionFvs) =
  let (subSet1, subSet2) =
        ifIsSubSetThenLeftUnionIsSubSet subSet isUnionFvs
      res1 = interpEnv env subSet1 e1
      res2 = interpEnv env subSet2 e2
  in res1 + res2
```

El evaluador tiene tipo `Ambiente fvsEnv ->IsSubSet fvs fvsEnv ->Exp fvs ->Nat`. Cada subexpresión tiene tipo `Exp fvs1` y `Exp fvs2` respectivamente. Para poder evaluar tales subexpresiones, como ya se tiene un ambiente del tipo `Ambiente fvsEnv`, es necesario tener una prueba de `IsSubSet fvs1 fvsEnv` y `IsSubSet fvs2 fvsEnv` para poder llamar al evaluador.

Por suerte eso es lo que efectivamente realiza esta función:

```
ifIsSubSetThenLeftUnionIsSubSet : DecEq lty =>
  {ls1, ls2, lsSub1, lsSub2 : List lty} -> IsSubSet ls1 ls2 ->
  IsLeftUnion_List lsSub1 lsSub2 ls1 ->
  (IsSubSet lsSub1 ls2, IsSubSet lsSub2 ls2)
```

Esta función toma una prueba de que `ls1` es subconjunto de `ls2`, una prueba de que `ls1` es el resultado de la unión por izquierda de `lsSub1` y `lsSub2`, y retorna las pruebas de que `lsSub1` y `lsSub2` son subconjuntos de `ls2`. Prueba que la unión por izquierda preserva el predicado de ser subconjunto de una lista para ambos componentes de la unión.

En el evaluador, se realiza la llamada de la siguiente forma:

```
(subSet1, subSet2) =
  ifIsSubSetThenLeftUnionIsSubSet subSet isUnionFvs
```

`isUnionFvs` tiene tipo `IsLeftUnion_List fvs1 fvs2 fvs` y `subSet` tipo `IsSubSet fvs fvsEnv`. Por lo tanto esta llamada obtiene las pruebas de `IsSubSet fvs1 fvsEnv` y `IsSubSet fvs2 fvsEnv` deseadas.

Con tales pruebas se pueden evaluar las subexpresiones de esta forma:

```
res1 = interpEnv env subSet1 e1
res2 = interpEnv env subSet2 e2
```

Al tener ya los resultados de tales evaluaciones, el resultado final es su simple suma:

```
res1 + res2
```

4.3.4. Expresiones con sustitución de variable

Una expresión donde se declara una sustitución de una variable `var` por un valor `n` en una subexpresión `e` es más difícil de evaluar que los demás casos, ya que requiere la manipulación del ambiente de variables, como se verá a continuación.

```
interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    with (isElem var fvsEnv)
  interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | Yes varInEnv =
    let
        (MkAmbiente recEnv) = env
        hasField = ifIsElemThenHasFieldNat varInEnv
        newRec = hUpdateAtLabel var n recEnv hasField
        newEnv = MkAmbiente newRec

        consSubSet =
          ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
        newSubSet = ifConsIsElemThenIsSubSet consSubSet varInEnv
    in interpEnv newEnv newSubSet e

  interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | No notVarInEnv =
    let (MkAmbiente recEnv) = env
        newRec = consRec var n recEnv
          {notElem = ifNotElemThenNotInNats notVarInEnv}
        newEnv = MkAmbiente newRec {ls = (var :: fvsEnv)}
        newSubSet =
          ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
    in interpEnv newEnv newSubSet e
```

El primer paso a tomar es saber si la variable `var` se encuentra en el ambiente o no. Es posible que la variable `var` ya tiene un valor `n2` asignado en el ambiente, pero si es así, entonces tal valor debe ser sustituido por `n` cuando se intente evaluar la subexpresión `e`. Si la variable no se encuentra en el ambiente, entonces debe ser agregado a tal al momento de evaluar la subexpresión `e`.

Veremos cómo se realiza esto en la implementación misma:

```
interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    with (isElem var fvsEnv)
```

El primer paso entonces consiste en tomar las variables del ambiente `fvs` y verificar si `var` pertenece a ellas con la llamada a `isElem var fvsEnv`. Esto trae dos casos posibles, uno donde pertenece y otro donde no.

Primero abarcaremos el caso donde no existe:

```
interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | No notVarInEnv =
    let (MkAmbiente recEnv) = env
```

Se obtiene la prueba `notVarInEnv : Not (Elem var fvsEnv)`, y luego se extrae el record de tipo `Record {lty=String} (AllNats fvs)` del ambiente.

```
newRec = consRec var n recEnv
  {notElem = ifNotElemThenNotInNats notVarInEnv}
```

Como la variable no pertenece al ambiente, entonces se agrega su etiqueta `var` y su valor `n` al record, extendiendo el record con `consRec`. Recordemos el tipo de `consRec`:

```
consRec : DecEq lty => {ts : LabelList lty} -> {t : Type} ->
  (l : lty) -> (val : t)->  Record ts ->
  {notElem : Not (ElemLabel l ts)} -> Record ((l,t) :: ts)
```

Para poder extender el record se necesita una prueba de `Not (ElemLabel l ts)`, es decir, que la etiqueta a agregar no exista en el record. Como este caso surge de tener una prueba de que la variable no está en el ambiente, es posible construir la prueba con la siguiente función:

```
ifNotElemThenNotInNats : Not (Elem x xs) ->
  Not (ElemLabel x (AllNats xs))
```

Recordemos que se tiene la prueba `notVarInEnv : Not (Elem var fvsEnv)`, pero para poder extender el record se necesita una prueba de `Not (ElemLabel var (AllNats fvsEnv))`. Esta función auxiliar simplemente realiza esa transformación, conociendo que si un elemento no pertenece a una lista como `['x, 'y']` entonces tampoco va a pertenecer a una donde se agrega el tipo `Nat`, como `[('x', Nat), ('y', Nat)]`.

Luego de extender el record, se obtiene el nuevo ambiente de forma simple:

```
newEnv = MkAmbiente newRec {ls = (var :: fvsEnv)}
```

Ahora que se tiene el nuevo ambiente, para poder evaluar la subexpresión en él, es necesario tener una prueba de que las variables libres de la subexpresión son un subconjunto de las del nuevo ambiente. Esto se realiza de esta forma:

```
newSubSet =
  ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
```

Conociendo que `subSet` es la prueba de `IsSubSet fvs fvsEnv`, y `delAt` de `DeleteLabelAtPred_List var fvsInner fvs`, entonces se necesita poder obtener la prueba de `IsSubSet fvsInner (var :: fvsEnv)`. Básicamente, si se agrega `var` a ambas listas la propidad de ser un subconjunto se preserva.

Esta prueba se obtiene con la siguiente función:

```
ifIsSubSetThenSoIfYouDeleteLabel :
  DeleteLabelAtPred_List l ls1 ls3 ->
  IsSubSet ls3 ls2 -> IsSubSet ls1 (l :: ls2)
```

Ahora se tiene el nuevo ambiente `Ambiente (var :: fvsEnv)`, se tiene la prueba de `IsSubSet fvsInner (var :: fvsEnv)` y la subexpresión `Exp fvsInner`. Con estos términos se puede evaluar tal subexpresión de esta forma, terminando la evaluación de la expresión en su conjunto:

```
interpEnv newEnv newSubSet e
```

Ahora solo queda realizar la evaluación cuando la variable `var` sí pertenece al ambiente, en este caso:

```
interpEnv {fvsEnv} env subSet (Let (var := n) e delAt)
    | Yes varInEnv =
    let
        (MkAmbiente recEnv) = env
```

Se tiene la prueba `varInEnv : Elem var fvsEnv`. Luego se extrae el record de tipo `Record {lty=String} (AllNats fvs)` del ambiente.

Como la variable pertenece al ambiente, entonces es necesario reemplazar su valor con `n`. Para ello, primero se debe obtener la prueba de que tal variable pertenece al record con tipo `Nat`:

```
hasField = ifIsElemThenHasFieldNat varInEnv
```

Para esto se utiliza la siguiente función auxiliar:

```
ifIsElemThenHasFieldNat : Elem l ls -> HasField l (AllNats ls) Nat
```

Esta función simplemente transforma una prueba a otra. `Elem l ls` es equivalente a `HasField l (AllNats ls) Nat`, porque sabemos que `AllNats` no agrega información a la lista más que todos tienen el tipo `Nat`.

Para actualizar el record, se utiliza la siguiente función:

```
hUpdateAtLabel : DecEq lty => (l : lty) ->
  ty -> Record ts -> HasField l ts ty -> Record ts
```

Esta función se vió en los ejemplos del capítulo anterior. Toma un record, una etiqueta de tal, una prueba de que esa etiqueta existe en el record y tiene un tipo `ty`, y actualiza el record con un valor del tipo `ty`. En este caso, se tiene una prueba de `HasField l (AllNats ls) Nat`, por lo que se puede actualizar el record pasando un nuevo valor de tipo `Nat` de la siguiente forma:

```
newRec = hUpdateAtLabel var n recEnv hasField
newEnv = MkAmbiente newRec
```

Al tener el record se puede crear el nuevo ambiente de tipo `Ambiente fvsEnv`, idéntico al anterior, solo con el valor `n` para la etiqueta `var`.

Ahora, al igual que el caso donde `var` no pertenecía al ambiente, es necesario construir la prueba de que la lista de variables libres de la subexpresión es un subconjunto de las del ambiente. Al igual que el caso anterior, se utilizará la misma función:

```
consSubSet =
  ifIsSubSetThenSoIfYouDeleteLabel delAt subSet {l = var}
```

Como en el caso anterior, esta llamada retorna una prueba de `IsSubSet fvsInner (var :: fvsEnv)`. Sin embargo, en este caso el ambiente no es de tipo `Ambiente (var :: fvsEnv)` sino que su tipo nunca cambió y sigue siendo `Ambiente fvsEnv`. En este caso es necesaria una prueba de `IsSubSet fvsInner fvsEnv`.

```
newSubSet = ifConsIsElemThenIsSubSet consSubSet varInEnv
```

Como se muestra arriba, tal prueba se consigue con la siguiente función:

```
ifConsIsElemThenIsSubSet : IsSubSet ls1 (l :: ls2) ->
    Elem l ls2 -> IsSubSet ls1 ls2
```

Esta función indica que si se tiene una prueba de que una lista `ls1` pertenece a `l :: ls2`, pero `l` ya pertenece a `ls2`, entonces es posible eliminarla y esto no altera la propiedad de ser subconjunto.

En el caso actual, al tener `IsSubSet fvsInner (var :: fvsEnv)` y `Elem var fvsEnv`, se sabe que `fvsInner` es subconjunto de `fvsEnv`, representado por la prueba de `IsSubSet fvsInner fvsEnv`.

Ahora se tiene el nuevo ambiente `Ambiente fvsEnv`, se tiene la prueba de `IsSubSet fvsInner fvsEnv` y la subexpresión `Exp fvsInner`. Con estos términos se puede evaluar tal subexpresión de esta forma, terminando la evaluación de la expresión en su conjunto:

```
interpEnv newEnv newSubSet e
```

Como se pudo evaluar la subexpresión para los dos casos (si `var` pertenece al ambiente o no), ya termina la evaluación de la expresión `let var := n in e`.

Capítulo 5

Conclusiones

El objetivo principal de este trabajo era el de poder implementar una biblioteca de records extensibles en un lenguaje con tipos dependientes, de tal forma que su desarrollo y uso sean más sencillos o adecuados gracias a tales tipos dependientes (y gracias al lenguaje utilizado, Idris en este caso). Por lo que se vió en este trabajo, este objetivo se cumplió satisfactoriamente. Es posible tener una solución al problema de records extensibles en un lenguaje como Idris, y éste presenta muchas ventajas sobre otras soluciones que vale la pena tener en cuenta.

A continuación se discutirán los problemas y facilidades que se encontraron en este trabajo

5.1. Viabilidad de implementar records extensibles con tipos dependientes

En este trabajo se vió que implementar records extensibles con tipos dependientes es viable. Una vez que se tienen claro los conceptos del lenguaje, el desarrollo y diseño de la solución se realizan con bastante simpleza y sencillez. No se utilizan técnicas muy avanzadas ni poco entendibles, y el diseño final queda entendible también.

Por ejemplo, la definición de records misma se realiza con solo dos líneas de código bastante entendibles:

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

Por otra parte, este trabajo constó en intentar traducir la implementación de HList (la biblioteca de Haskell) a Idris. En ese aspecto, la traducción final resultó muy similar a la de HList, y no se encontraron problemas mayores en traducir conceptos, tipos y funciones de uno a otro. El diseño actual, basado en HList, permite tener una solución adecuada de records extensibles en Idris que cumple con todos los requisitos de éste.

El objetivo de hacer la biblioteca amigable para el usuario también se cumplió bastante. Con esta biblioteca crear records y manipularlos se puede realizar sin tener que utilizar funcionalidades avanzadas del lenguaje, ni recurrir a técnicas complejas. Con funciones como `consRecAuto` y otras se pueden crear records de

forma sencilla, garantizando que el record final cumpla con las propiedades deseadas (e.g que no contenga etiquetas repetidas).

Algunos aspectos que no resultan amigables al usuario es cuando se necesita manipular el record de forma genérica. Es decir, si el usuario tiene un record en particular, como `{ 'x' : 10, 'y' : 15 }`, puede manipularlo de forma sencilla. Sin embargo, si el usuario tiene un record como argumento o resultado de una función, para poder manipularlo va a necesitar manipular las pruebas de sus propiedades de forma manual. Como se vió en el caso de estudio, en la implementación del evaluador de expresiones era necesario que el usuario genere las pruebas de `HasField`, `Not (ElemLabel l ls)`, etc. A diferencia de otros lenguajes y soluciones de records extensibles, esto le agrega un nivel de complejidad mayor al uso de ellos, ya que el usuario necesita estar familiarizado con esas definiciones de propiedades y debe estar familiarizado con Idris para saber cómo crear sus pruebas y manipularlas.

En términos generales, se vió que esta solución de records extensibles tiene ventajas sobre otras realizadas en otros lenguajes. Comparándola con HList de Haskell, esta solución tiene el mismo poder de expresión de ella (al ser una traducción casi directa). Comparándola con otros lenguajes como Elm y Purescript, como se vió en el capítulo *'Records extensibles en Idris'* esta solución permite modelar las mismas funcionalidades de ellos, pero a su vez le da una mayor cantidad de funcionalidades y flexibilidad al usuario.

5.2. Desarrollo en Idris y con tipos dependientes

En esta sección se describirán algunos problemas que se encontraron en la forma de desarrollar en Idris y utilizando tipos dependientes.

El primer aspecto importante que se vió, es que el trabajo mayor de desarrollo no se encontraba en las definiciones de tipos y funciones más importantes, sino en el desarrollo de las pruebas de teoremas y lemas auxiliares o intermedios. En este trabajo se mencionaron varios teoremas sin mostrar su implementación (que se encuentra en el apéndice), pero varios de esos teoremas, para poder ser probados, necesitan de varios sub-lemas, necesitan realizar muchos análisis de casos, y en general resulta más difícil solucionarlos, ya que se entra en el territorio de pruebas formales e inductivas.

A su vez, cuantos más predicados y funciones se agregan, se encontró que la cantidad de lemas y teoremas necesarios aumenta considerablemente. Por ejemplo, al agregar predicados como `IsLeftUnion` o `HasField`, o funciones como `(++)` (append), es necesario crear teoremas que relacionen esos predicados y funciones con otras previamente creadas. Por ejemplo:

```
ifNotInEitherThenNotInAppend : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  Not (ElemLabel l ts1) -> Not (ElemLabel l ts2) ->
  Not (ElemLabel l (ts1 ++ ts2))
```

En el caso de estudio se definió el predicado `IsSubSet` para indicar que una lista es un subconjunto de otra. En este caso también era necesario tener que crear varios teoremas que relacionaran este predicado con los otros utilizados en la biblioteca. Por ejemplo:

```
ifIsSubSetOfEachThenIsSoAppend : DecEq lty =>
  {ls1, ls2, ls3 : List lty} ->
  IsSubSet ls1 ls3 -> IsSubSet ls2 ls3 ->
  IsSubSet (ls1 ++ ls2) ls3
```

Esto resulta en varios problemas:

- El trabajo del desarrollador de la biblioteca se multiplica cada vez que quiera agregar nuevas funcionalidades sobre los records, o cuando quiera agregar nuevos predicados o funciones sobre ellos. Por ejemplo, si se quisiera agregar la funcionalidad `unzipRecord` de HList, se deberían agregar nuevos predicados y funciones para las listas de campos. Esto requeriría agregar varios teoremas de cómo se relacionan estos predicados y funciones con `ElemLabel`, `IsLeftUnion`, `IsLabelSet`, `IsProjectLeft`, entre otros.
- El usuario va a tener que implementar sus propios teoremas para cualquier predicado nuevo que él cree. En el caso de estudio, el nuevo predicado fue `IsSubSet`. Al crear este nuevo predicado era necesario agregar nuevos teoremas, como el teorema `ifIsSubSetOfEachThenIsSoAppend` descrito anteriormente. Esto puede resultar muy costoso para el usuario.

En relación a los teoremas y predicados, la biblioteca en sí debería proporcionar todos los predicados y teoremas que pueda. Para poder tener las definiciones de los tipos y predicados encapsulados, sin que sea necesario que el usuario tenga que conocer sus implementaciones, no se debe poder esperar que el usuario mismo cree teoremas sobre estos predicados de tal forma que tenga que operar con su definición o implementación explícitamente. Esto requiere de un mayor trabajo para el desarrollador de la biblioteca, ya que debe crear teoremas para cualquier posible caso y uso de sus predicados y funciones, todo con el afán de facilitarle su uso al usuario. De lo contrario, los usuarios mismos tienen que conocer la implementación interna de la biblioteca y crear sus propios teoremas, lo cual resulta inviable.

En el desarrollo de este trabajo hubo algunos problemas de diseño que surgieron a causa de los tipos dependientes. En algunos momentos hubo algunos predicados sobre listas que tuvieron código duplicado. Los predicados `IsLeftUnion` trabajan sobre listas de tipo `LabelList lty` mientras que predicados `IsLeftUnion_List` trabajan sobre listas de tipo `List lty`. Parametrizar tales predicados por cualquier tipo de lista no es trivial y requiere de un rediseño de la solución. Varios de estos problemas surgen en una etapa avanzada del desarrollo (e.j el problema de `IsLeftUnion` descrito anteriormente surgió al momento de implementar el caso de estudio), lo cual hace más costoso su cambio.

Otro problema que surgió en el desarrollo es que la lectura del código no proporciona la información necesaria al programador. Las pruebas de los teoremas y lemas se realiza paso a paso. En cada rama del código, se tienen términos de prueba y valores con distintos tipos complejos, y se deben combinar ellos para poder obtener la prueba final. Muchos de estos términos son implícitos y no aparecen en el código (por ejemplo al utilizar la sintaxis `{}` de Idris). Una vez finalizada la prueba, es imposible conocer los tipos y el estado de esos términos en tales ramas, ya que o son términos implícitos o sus tipos no se muestran explícitamente en el código. Esto dificulta no solo la tarea de lectura y entendimiento de las pruebas, sino que si se realiza un cambio en la definición de un predicado o función, al desarrollador le resulta difícil entender donde tiene que realizar los cambios correspondientes para adaptarse a ese cambio de definición.

Sin ser por los problemas descritos, el desarrollo en Idris es bastante placentero y sencillo. Al tener tipos más ricos, la implementación de los teoremas y funciones se simplifica bastante con la ayuda del compilador, que previene que uno tome en cuenta casos inválidos (como `Elem x []`). Los tipos dependientes ayudan mucho en este aspecto, ya que cuanta más información se tenga en el tipo que permita restringir los posibles estados de sus valores, el programador debe realizar menos análisis de casos, y se reduce la chance de introducir defectos por olvidarse de manejar alguno de tales casos, o por manejarlos de forma errónea.

5.3. 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.

5.3.1. 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
```

5.3.2. 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
```

5.3.3. 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 6

Trabajo a futuro

En este trabajo se encontraron varias tareas para mejorar esta implementación de records extensibles.

- Una posible tarea a futuro es terminar de implementar las demás funciones y predicados de *HList* de Haskell que faltan implementar.
- En el diseño de la biblioteca y del caso de estudio, hubo varios predicados y funciones duplicados para `List (lty, Type)` y `List String`. Predicados como `IsLeftUnion`, `DeleteLabelAt`, y muchas otras. Una mejora es crear predicados y funciones parametrizadas por `List a`, con una restricción sobre este `a` de que contenga una etiqueta o valor String. De este modo no se repite código ni se realiza más esfuerzo en probar todos los lemas para cada predicado en particular.
- Todavia existen muchos lemas y teoremas que relacionan todos los predicados y tipos creados que podrían ser muy útiles. Es fundamental que la biblioteca contenga estos teoremas para que el programador que la use no tenga que probarlos por sí solo cada vez (lo cual va a requerir que el usuario de esta biblioteca conozca su comportamiento interno, que es necesario para probar los teoremas).
- Para algunos predicados se crearon funciones que computan su mismo resultado. Por ejemplo, para el predicado `IsLeftUnion` se implementó la función `leftUnion`. Un buen trabajo a futuro sería implementar las funciones análogas de todos los predicados definidos, y definir todos los posibles teoremas y lemas interesantes entre esas funciones y los distintos predicados.
- La implementación actual funciona sobre listas ordenadas. Esto hace que `Record [('A', Nat), ('B', Nat)]` sea distinto de `Record [('B', Nat), ('A', Nat)]`, cuando en realidad tal orden no es necesario para utilizar las funciones definidas para records. Un posible trabajo es parametrizar el tipo del record por un tipo abstracto. Este tipo abstracto debe funcionar como una colección de etiquetas y tipos, y debe permitir operaciones como `head`, `index`, `cons`, `nil`, entre otras. Al funcionar como tipo abstracto se podría utilizar cualquier tipo que cumpla esas propiedades, eligiendo el que se adecúe a la situación del usuario. Se podría utilizar árboles binarios, árboles Red-Black, Fingertrees, etc.

- Actualmente los records se implementaron siguiendo los pasos de *HList* de Haskell. Sin embargo, pueden existir otras implementaciones que sean mejores en otras situaciones. Una buena tarea a futuro es diseñar la biblioteca utilizando el record como tipo abstracto, tal que la implementación misma del record pueda variar sin que los usuarios de la biblioteca sean afectados.
- Como se vió en la introducción, los conceptos visitados en este trabajo aplican tanto a *Agda* como a *Idris*. Sería interesante estudiar replicar esta implementación en Agda.
- Se puede mejorar los mensajes de error en la generación automática de errores. Cuando se utiliza `TypeOrUnit`, si el compilador no puede generar el predicado esperado, entonces se muestra un error de unificación de `TypeOrUnit` con el tipo esperado. Se puede investigar una nueva forma de utilizar `TypeOrUnit` para mostrar mejores mensajes de error. Se debería poder asignar un mensaje de error a cada aplicación de `TypeOrUnit`, como por ejemplo el mensaje *"El elemento a agregar ya existe"* al llamar a `consRec`.

Capítulo 7

Apéndice

7.1. Código fuente

PENDIENTE

Bibliografía

[1] *An overview of Hugs extensions - Extensible records: Trex*. 1999. URL: https://www.haskell.org/hugs/pages/hugsman/exts.html#sect7.2 (visitado 07-02-2017).

[2] Julian K. Arni. *Bookkeeper*. 2015. URL: https://github.com/turingjump/bookkeeper (visitado 09-02-2017).

[3] Edwin Brady. «Idris, a general-purpose dependently typed programming language: Design and implementation». En: *Journal of Functional Programming* 23 (sep. de 2013), págs. 552-593. ISSN: 1469-7653. DOI: 10.1017/s095679681300018x. URL: http://dx.doi.org/10.1017/s095679681300018x.

[4] Luca Cardelli y John C. Mitchell. «Operations on Records». En: *Proceedings of the Fifth International Conference on Mathematical Foundations of Programming Semantics*. New Orleans, Louisiana, USA: Springer-Verlag New York, Inc., 1990, págs. 22-52. ISBN: 0-387-97375-3. URL: http://dl.acm.org/citation.cfm?id=101514.101515.

[5] Chih-Mao Chen. *The rawr package*. 2016. URL: https://hackage.haskell.org/package/rawr (visitado 09-02-2017).

[6] Evan Czaplicki y Stephen Chong. «Asynchronous Functional Reactive Programming for GUIs». En: *SIGPLAN Not.* 48.6 (jun. de 2013), págs. 411-422. ISSN: 0362-1340. DOI: 10.1145/2499370.2462161. URL: http://doi.acm.org/10.1145/2499370.2462161.

[7] Chris Done. *The labels package*. 2016. URL: https://hackage.haskell.org/package/labels (visitado 09-02-2017).

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

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

[10] Julian Fleischer. *The named-records package*. 2013. URL: https://hackage.haskell.org/package/named-records (visitado 09-02-2017).

[11] Nicolas Frisby. *The ruin package*. 2016. URL: https://hackage.haskell.org/package/ruin (visitado 09-02-2017).

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

[13] William A. Howard. «The formulas-as-types notion of construction». En: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. por J. P. Seldin y J. R. Hindley. Academic Press, 1980, págs. 479-490.

[14] 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.

[15] Wolfgang Jeltsch. *The records package*. 2012. URL: https://hackage.haskell.org/package/records (visitado 09-02-2017).

[16] Mark P. Jones y Simon Peyton Jones. *Lightweight Extensible Records for Haskell*. 1999.

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

[18] 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.

[19] Edward A. Kmett. *The tagged package*. 2010. URL: https://hackage.haskell.org/package/tagged (visitado 10-02-2017).

[20] 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.

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

[22] Ulf Norell. «Dependently Typed Programming in Agda». En: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA: ACM, 2009, págs. 1-2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862. URL: http://doi.acm.org/10.1145/1481861.1481862.

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

[24] *Purescript by Example*. 2016. URL: https://leanpub.com/purescript (visitado 31-01-2017).

[25] Jonathan Sterling. *The vinyl package*. 2012. URL: https://hackage.haskell.org/package/vinyl (visitado 09-02-2017).