Universidad de la República, Uruguay

Proyecto de Grado

Desarrollo de records extensibles en lenguajes con tipos dependientes

Gonzalo Waszczuk

Supervisado por
Marcos Viera
Alberto Pardo

Resumen

Los records extensibles son estructuras de datos que permiten asociar valores de distintos tipos a etiquetas para poder ser accedidos mediante ellas, al igual que permiten extender esa estructura con nuevos campos de forma dinámica. Existen varias soluciones de records extensibles en varios lenguajes, sean como primitivas de lenguaje o como bibliotecas de usuario, cada una con sus ventajas y desventajas. Una de ellas es la biblioteca *HList* de Haskell, que utiliza listas heterogéneas para definir tales records. En este trabajo se presentará una solución de records extensibles en un lenguaje fuertemente tipado con tipos dependientes llamado Idris, basándose en la biblioteca *HList*. Este trabajo explorará las propiedades de los tipos dependientes y el desarrollo en Idris, y demostrará que es posible realizar una implementación de records extensibles en base a ellos, con resultados interesantes. Se presentarán los beneficios de esta solución en el contexto de un lenguaje de expresiones aritméticas definido como un *embedded Domain Specific Language* en Idris.

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 programación con tipado estático 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 vez definidos los records 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 records extensibles, 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. A su vez, se pueden acceder a los campos `edad` y `nombre` como se podía hacerlo antes de extender el record.

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 los records extensibles, y no es posible implementarlos en términos de 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, por ejemplo, como un valor de tipo `String`, quedando el tipo final `Persona + { 'apellido' :: String }`. Al ser un valor, se podría, por ejemplo, guardarlo en una variable y utilizarlo de la siguiente manera:

```
miCampo = String
miCampo = 'apellido'

pExtensible :: Persona + { miCampo :: 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`.

Para conocer mejor el uso de tipos dependientes, veamos la definición del tipo *vector* en *Idris* [3], que representa listas cuyo largo es anotado en su tipo:

```
data Vect : Nat -> Type -> Type where
  [] : Vect 0 A
  (::) : (x : A) -> (xd : 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 función:

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

Esta función obtiene el primer valor de un vector. Es una función total, ya que restringe su uso solamente a vectores que tengan un largo mayor a 0, por lo que el vector al que se le aplique esta función 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 typechecker no va a poder unificar el tipo `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 typechecking).

En relación a records extensibles, los tipos dependientes hacen posible que dentro del tipo del record puedan existir valores para poder ser manipulados, como podría ser la lista de etiquetas del record. Como esta información se encuentra en el tipo del record, es posible definir tipos y funciones que accedan a ella y la manipulen para poder definir todas las funcionalidades de records extensibles.

Otra propiedad de los tipos dependientes es que permiten definir predicados como tipos. Es posible definir predicados como tipos, y probar tales predicados 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 decidió utilizar el lenguaje de programación *Idris* [3] para llevar a cabo la investigación del uso de tipos dependientes aplicados a la definición de records extensibles. Idris es un lenguaje de programación funcional con tipos dependientes, con sintaxis similar a Haskell.

Otro lenguaje que cumple similares requisitos es *Agda* [22]. Sin embargo, se decidió utilizar Idris para investigar las funcionalidades de este nuevo lenguaje. Las conclusiones obtenidas en este trabajo podrían ser aplicadas a *Agda* también.

La versión de Idris utilizada en este trabajo es la 0.12.0

1.4. Guía de trabajo

A continuación se describe la organización del resto del documento:

En el capítulo 2 se muestran varias implementaciones de records extensibles en otros lenguajes, mostrando sus beneficios e incovenientes. También se muestra una implementación específica de records extensibles en Haskell.

En el capítulo 3 se presenta 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. También se muestran los problemas encontrados en este trabajo y sus respectivas soluciones.

En el capítulo 4 se presenta 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 de records extensibles y qué se puede llegar a hacer con ella.

En el capítulo 5 se toman las experiencias del desarrollo del trabajo y el caso de estudio y se analizan los problemas encontrados, el aporte que puede llegar a tener este trabajo sobre el problema de records extensibles, entre otras cosas.

En el capítulo 6 se indican aspectos de diseño e implementación que pueden realizarse sobre este trabajo para mejorarlo, como posibles tareas que surgieron y se pueden llevar a cabo.

En el apéndice se encuentran recursos adicionales, como el código fuente de este trabajo.

Capítulo 2

Estado del arte

En este capítulo se presetan varias implementaciones de records extensibles en varios lenguajes. El trabajo se enfoca en lenguajes fuertemente tipados. Existen implementaciones de records extensibles en lenguajes con tipado dinámico, pero caen fuera del alcance de este trabajo.

En términos generales, las implementaciones de records extensibles se dividen según si son proporcionadas por el lenguaje como primitivas, o si son proporcionadas 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 se presenta más adelante.

2.1. Records extensibles como primitivos

Algunos lenguajes 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 }
```

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 tenga como mínimo unos determinados campos, pero el resto de éstos puede variar. Su uso sería el siguiente:

```
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
  }
```

En el ejemplo de arriba, el record `lady` es definido extendiendo `Named` con otro campo adicional, sin necesidad de definir un tipo nuevo.

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 [20]. En dicho trabajo Leijen describe un sistema de tipos donde los records extensibles son una nueva extensión del lenguaje, y permite que éstos puedan contener etiquetas repetidas, asignando un *alcance* a cada etiqueta. Cada acceso a un elemento del record mediante una etiqueta simplemente accede al primer elemento de éste con esa etiqueta. La sintaxis y operaciones sobre records que se definen en este paper es la utilizada en Elm y Purescript, como por ejemplo `{ l = e | r }`, que extiende un record `r` con la etiqueta `l` y el valor `e`. Este trabajo implementa *row polymorphism* al designar un *kind* (tipo de tipos) llamado *row* y permitiendo definir tipos de records parametrizados por tales rows.

Existen otras propuestas de sistemas de tipos con soporte para records extensibles. En [21], Leijen describe un sistema de tipos con etiquetas como *first-class citizens* del lenguaje para permitir construir records extensibles de forma expresiva.

En [12], Gaster y Jones también describen un sistema de tipos con soporte para records extensibles. Ambos trabajos se basan en extensiones del sistema de tipos de Haskell y ML, incluyendo inferencia de tipos. El trabajo de Gaster y Jones fue utilizado para la extensión Trex de records extensibles de Hugs98 [1] (Hugs98 es un intérprete de Haskell que desafortunadamente ha sido descontinuado).

Otra propuesta es la de [4]. Esta propuesta desarrolla una teoría de records extensibles, formalizando un sistema de tipos para soportarlos. Otra propuesta es [16], de Jones y 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 records extensibles 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 [16], los autores describen un problema de su extensión de records, donde al tener un nuevo kind (tipo de tipos) sobre records, existen ambigüedades 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 requerir un rediseño de muchas partes de éste 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*. En algunas propuestas vistas, el lenguaje requiere de sintaxis especial para el manejo de records (como el kind `ROW` en Haskell). Esta falta de soporte limita la expresividad de tales records, ya que divide el lenguaje en dos, separando la manipulación del record extensible de la manipulación de otros valores del lenguaje. Esta separación evita que, por ejemplo, se puedan desarrollar funcionalidades parametrizables por ambos, como por ejemplo, desarrollar una función que se pueda parametrizar tanto por un valor común (como uno de tipo `Int`) como por un record extensible.

Este trabajo se enfoca en el abordaje a 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, ya que puede elegir la implementación de records extensibles que más desee y le resulta mejor para su situación.

2.2. Records extensibles como bibliotecas de usuario

Las bibliotecas de usuario son componentes de un lenguaje de programación que están escritas en ese mismo lenguaje, proveyendo una funcionalidad al usuario que utiliza este componente. 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 [14], 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].

Otras propuestas se basan en utilizar extensiones de Haskell para poder definir los records. A modo de ejemplo, *rawr* [5] utiliza *type families* y *type-level lists* para definir sus records; *Vinyl* [25] también utiliza type-level lists, pero define un record como un *GADT* (Generalized Abstract Data Type); *ruin* [11] utiliza *Template Haskell* y metaprogramming para crear instancias de typeclasses 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 )

foo :: Foo
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). Una diferencia a notar, es que la definición del tipo `Foo` se utilizaron strings, como `"a"`, mientras que en la definición de un valor de ese tipo se utilizaron identificadores, como `#a`. Esto sucede porque los records en *rawr* hacen uso de *type-level literals*, una funcionalidad del compilador GHC que permite utilizar strings literales en un tipo (llamados *Symbols*). Por lo que `"a"` en `R ( "a" := Int)` es un string promovido a un tipo.

Para poder utilizar un valor literal promovido, en GHC se utiliza el tipo `Proxy t`, el cual solo mantiene una referencia al tipo `t`. En este caso, `#a` es un valor de tipo `Proxy "a"`.

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 )
```

El operador `(:*:)` realiza una unión de dos records. Si uno quiere extender un record `foo` con una etiqueta `bar` y un valor `x`, se puede encapsular los dos últimos en un record y realizar la unión del primero con este último:

```
extend foo bar x = foo :*: R ( bar := x)
```

Para poder obtener un elemento de un record, se puede utilizar su etiqueta directamente como función de acceso:

```
rec = R ( #foo := True)
#foo rec
--> True
```

Entre las muchas propuestas y alternativas, el presente trabajo se basó en la biblioteca HList de Haskell [17]. HList muestra las problemáticas de definir records extensibles, a la vez que soluciones a ellas. También tiene varias propiedades que son deseadas tener en un diseño de records extensibles (como flexibilidad para agregar nuevas operaciones sobre records). El objetivo del presente trabajo es trasladar las ideas de HList a un contexto de tipos dependientes en Idris y hacer uso de sus buenas propiedades.

Uno de los mecanismos utilizados por HList y la mayoría de las bibliotecas previamente descritas son las *listas heterogéneas*, que se presentan en la siguiente sección. Bibliotecas como HList y Records [15] 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 de esa manera es posible definir records extensibles de una 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 describien las listas heterogéneas, luego se muestra cómo se implementan en la biblioteca HList de Haskell, y al final se presenta cómo son implementadas en Idris.

2.3. Listas heterogéneas

El concepto de lista heterogénea (o *HList*) surge en oposición al tipo de listas usualmente utilizado en lenguajes tipados: listas homogéneas. Las listas homogéneas 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 `[Int]` en Haskell.

Las listas heterogéneas, sin embargo, permiten almacenar elementos de distintos tipos. Estos tipos pueden no tener una relación entre ellos.

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 intérprete (o compilador) 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 le puede agregar un entero, un float y un texto:

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

En lenguajes fuertemente tipados este tipo de lista es más difícil de construir. Tales listas heterogéneas 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. No todos los lenguajes fuertemente tipados lo permiten.

En sistemas de tipos más avanzados es posible incluir la 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 [18], por Kiselyov, Lämmel y Schupke. Esta propuesta está implementada en el paquete *hlist* de Hackage [17]. Su implementación es muy similar a la de la biblioteca *records* descrita anteriormente.

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`, que toma una lista de tipos como parámetro, definiendo un caso cuando una lista de tipos es vacía y otro caso cuando no lo es.

Data families es una extensión de Haskell, junto a *type synonym families* (juntas denominadas *type families*), que permiten realizar overloading sobre tipos paramétricos. Cada type family se parametriza sobre un tipo y permite generar representaciones

distintas para distintas instancias de ese tipo. Si se tiene `MyType a` como type family, entonces la representación de ese tipo va a ser distinta para `MyType Int` que para `MyType String`.

En una *data family*, para cada instancia del tipo paramétrico en particular se definen constructores distintos. Como ejemplo:

```
data family MyData a
data instance MyData Int =
  MyDataC1 Int | MyDataC2 String
data instance MyData String = MyDataC3 [Int]
```

Si en algún momento se tiene `MyData Int`, entonces es posible realizar pattern matching sobre él con los constructores `MyDataC1` y `MyDataC2`, pero *no* con `MyDataC3`. Si se tiene `MyData String` ocurre lo contrario, se puede realizar pattern matching con el constructor `MyDataC3` pero no con los demás. Si se tiene `MyData a` parametrizado por un `a` arbitrario, no se puede realizar pattern matching hasta conocer más sobre el tipo `a`.

A diferencia de *data families*, las instancias de un *type synonym family* indican que, para cada instancia del tipo paramétrico en particular, su representación es un sinónimo de tipo distinto. Como ejemplo:

```
type family MySyn a
type instance MySyn Int = Maybe Int
type instance MySyn String = [Int]
```

En este caso, cada vez que se tiene `MySyn Int`, éste funciona como un alias para `Maybe Int`. Cada vez que se tiene `MySyn String`, éste funciona como un alias de `[Int]`. Si se tiene `MySyn a` parametrizado por un `a` arbitrario, no se puede conocer su sinónimo hasta tener más información sobre el tipo `a`.

Para el caso de `HList t`, ésta se define como un data family de dos constructores, `HNil` y `HCons`, donde `HNil` solo puede aplicarse cuando la lista `t` es vacía, y `HCons` cuando ésta no es vacía. Esto permite, por ejemplo, construir el siguiente valor:

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

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.

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

```
hLength :: HList l -> Proxy (HLength l)
hLength _ = Proxy

data Proxy a = Proxy

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 definición de una función y que el typechecker decida cuál de las instancias anteriores debe llamar, culminando en un valor del kind `HNat`.

Luego, la función `hLength` toma una lista heterogénea, calcula su largo con `HLength`, y lo retorna en un tipo `Proxy a` (un *phantom type* que solo mantiene una referencia a `n`, el largo de la lista, en su tipo).

Un ejemplo de uso de tal función sería el siguiente:

```
hLength (10 `HCons` ("Texto" `HCons` HNil)) =
  Proxy :: Proxy (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. La intención es poder manipular listas heterogéneas con la misma facilidad que uno lo hace con listas homogéneas. Esto no ocurre en el caso de HList en Haskell, ya que el tipo de HList es definido con data families, y las funciones sobre HList requieren ser definidas con type families y typeclasses. 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, Idris maneja tipos dependientes. Esto significa que cualquier tipo puede estar parametrizado por un valor, y en tal caso el tipo es un ciudadano de primera clase que puede ser utilizado como cualquier otro elemento del lenguaje. Por lo tanto, para Idris no hay diferencia en el trato de un tipo simple como `String` o un tipo complejo con tipos dependientes como `Vect 2 Nat`.

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] :: Nil :
  HList [Nat, String, [Nat]]
```

El tipo `HList` se define como un tipo indexado por una lista de tipos (e.j `[Nat, String]`). 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. Por ejemplo, al agregar `23 : Nat` se guarda `23` en la lista y `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, el hecho de que los tipos dependientes son ciudadanos de primera clase 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 por lo tanto solo opera a nivel de *tipos*. En cambio,

el uso de HList en Idris no es distinto del uso de listas comunes (`List`), y por lo tanto pueden ser tratadas de la misma forma tanto por quienes crean la biblioteca que por quienes la utilizan.

A continuación se presenta 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 de `HRLabelSet` 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 typeclass, la lista necesita cumplir el predicado `HAllTaggedLV` y `HLabelSet`. Para esto la lista debe contener valores de 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].

Estas etiquetas no deben repetirse, y para ello la lista debe poder tener una instancia de `HLabelSet (LabelsOf ps)`. `LabelsOf` es una type family que toma una lista de elementos de tipo `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 types* de `Tagged` y los retorna como etiquetas `Label l`. Como en algunas partes de la implementación se permite que la lista heterogénea tenga solo etiquetas (sin valores), la lista original también puede tener un valor del tipo `Label l`. El tipo `Label l` representa una etiqueta a nivel de tipos para poder identificar los campos del record. Su definición es idéntica a la de `Proxy` descrita en secciones anteriores, y es la siguiente:

```
data Label l = Label
```

`HLabelSet` es una typeclass 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 paso inductivo. Hace uso de predicados de igualdad de tipos para poder realizarlo. Su definición es la siguiente:

```
class Fail
data DuplicatedLabel l

class HLabelSet ls
instance HLabelSet '[]
instance HLabelSet '[x]
instance (HEqK l1 l2 leq,
  HLabelSet' l1 l2 leq r,
  ) => HLabelSet (l1 ': l2 ': r)

class HLabelSet' l1 l2 (leq :: Bool) r
instance (HLabelSet (l2 ': r),
  HLabelSet (l1 ': r),
  ) => HLabelSet' l1 l2 False r
instance (Fail (DuplicatedLabel l1)) =>
  HLabelSet' l1 l2 True r
```

Como casos base, se define que `'[]` y `'[x]` son conjuntos. Para el caso recursivo, se hace uso de una typeclass auxiliar `HLabelSet' l1 l2 (leq :: Bool) r`. Se fuerza a que `l1` y `l2` sean distintos utilizando la igualdad a nivel de tipos `HEqK l1 l2 leq` y la segunda instancia de `HLabelSet'`. Si `l1` y `l2` fueran iguales, entonces `HEqK l1 l2 True` sería la instancia obtenida, la cual unificaría con `HLabelSet' l1 l2 True r`. En este caso, esto fallaría en tiempo de compilación ya que utilizaría la segunda instancia de `HLabelSet'`, la cual necesita de una instancia de `Fail (DuplicatedLabel l1)`, que no existe porque la typeclass `Fail` se define sin instancias (es una typeclass para indicar casos de error). Esto fuerza a que `l1` y `l2` sean distintos. En tal caso, si se cumple que `(l1 ': r)` y `(l2 ': r)` son conjuntos sin elementos repetidos, se puede concluir que `(l1 ': l2 ': r)` no tiene elementos repetidos, representado por la instancia `HLabelSet (l1 ': l2 ': r)`.

La biblioteca también proporciona otra forma de generar records utilizando etiquetas y operadores especiales. A continuación mostraremos un ejemplo de su uso, adaptando el ejemplo visto en el paper de HList :

Teniendo las etiquetas `clave`, `nombre`, y `edad` previamente definidas, la biblioteca permite definir registros de la siguiente manera:

```
persona = clave .=. 3
  .*. nombre .=. "Juan"
  .*. edad .=. 27
  .*. emptyRecord
```

Los campos del record se definen utilizando el siguiente operador:

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

A su vez, el operador `(.*.)` permite extender un record con un nuevo campo. Su tipo es el siguiente:

```
(.*.) :: HExtend e l => e -> l -> HExtendR e l
```

`HExtendR e l` es una type family que representa el resultado final de extender el record `e` con el campo `l`. A su vez, la typeclass `HExtend` funciona como un predicado que indica que es posible extender `e` con `l`. Su implementación no se mostrará, pero cabe notar que la misma termina realizando un llamado a la función `mkRecord` vista anteriormente.

En HList, dos etiquetas `Label (Lbl x1 n1 desc)` y `Label (Lbl x2 n2 desc)` son iguales si sus namespaces y posiciones son idénticas. Es decir, si `x1 = x2` y `n1 = n2`. Por este motivo el chequeo de etiquetas repetidas y su búsqueda se realiza tomando valores de `Label` y verificando su posición en la lista de etiquetas creadas con `firstLabel` y `nextLabel`. Los textos en sí (como el string `'clave'`) no son utilizados.

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
```

Esta typeclass hace uso de *dependencias funcionales*. Las dependencias funcionales son una extensión de Haskell que permiten definir typeclasses donde algunos de sus argumentos determinan de forma única el resto. En el caso anterior, la typeclass `HasField` tiene 3 dependencias, `l`, `r` y `v`. Sin embargo, tiene una dependencia `l r ->v` que indica que solo puede existir una única instancia de `HasField` para cada par de `l` y `r`. Su concepto es análogo al de dependencias funcionales en bases de datos relacionales.

La función `hLookupByLabel` toma una etiqueta `Label l` y verifica si ésta existe en el record `r`. Cuando encuentra la etiqueta, retorna el valor que se encuentra en la estructura `Tagged`.

Para este trabajo se tomó como motivación esta implementación de records extensibles, realizando una traducción a Idris de cada función, tipo y algoritmo. Una diferencia de este trabajo con HList es que en Idris se realizó la comparación de etiquetas comparando por el texto de las mismas, mientras que en HList éstas se comparan utilizando su posición y namespace.

En el siguiente capítulo se muestra la implementación realizada en este trabajo.

Capítulo 3

Records extensibles en Idris

En este capítulo se describe cómo se implementaron los records extensibles en Idris. Se comienza mostrando ejemplos de creación y uso de records. Luego se explica el diseño de los records y cuáles funcionalidades de Idris lo hicieron posible. En otra sección se muestra la implementación de algunas de las funciones sobre records extensibles. No se muestran sus implementaciones completas, sino que se describen los rasgos más importantes de la implementación, dejando los detalles y funciones auxiliares para ver en el apéndice. El capítulo termina comparando esta solución de records extensibles con otras vistas en el capítulo 2.

3.1. Introducción a 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 el capítulo anterior:

```
persona = clave .=. 3
  .*. nombre .=. "Juan"
  .*. edad .=. 27
  .*. emptyRecord
```

Este caso puede expresarse en Idris de la siguiente 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 los de HList. La diferencia entre ambos es que Haskell infiere el tipo de `persona` pero Idris no.

```
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. Está formada por tuplas, donde

el primer valor de la tupla es la etiqueta del campo, y el segundo valor es el tipo del campo. Aquí se aprecia el uso de tipos dependientes, ya que el tipo `Record` está indexado por un valor (una lista en particular).

En Idris el tipo de `Record` es `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)] : Type`.

Dentro del tipo del record se encuentra toda la información necesaria, ya que se encuentran etiquetas y el tipo de cada campo. No es necesario definir valores externos al record como las etiquetas `clave` de HList (de tipo `Label`), sino que se pueden usar simplemente tipos como `String` para definir las etiquetas.

Otra pieza fundamental en este trabajo es `consRecAuto`, que es la función que permite tomar un record y extenderlo con otro campo. Consideremos el siguiente ejemplo:

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

Una vez que tenemos este record podemos ahora extenderlo con un campo más:

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

El nuevo record, además de tener el campo adicional, tiene anotado en su tipo la tupla `("Clave", Nat)` correspondiente a ese nuevo campo. El tipo de `consRecAuto` es el siguiente:

```
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 cuáles otros argumentos fueron pasados a la función, aunque a veces no puede hacerlo y es necesario pasarlos explícitamente. En este caso, `consRecAuto "Clave" 3 persona2` es equivalente a `consRecAuto {A = Nat} "Clave" 3 persona2`.

Inductivamente, se puede seguir aplicando el razonamiento anterior para deducir el tipo de `emptyRec`:

```
emptyRec : Record []
```

Además de la operación para la extensión de records, en este trabajo se definieron operaciones para la creación, manipulación y lookup de records. A modo de ejemplo, se tienen estos dos records, que representan distintos atributos de una persona:

```
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" 2 $
  consRecAuto "Edad" 34 $
```

```
  emptyRec
-- { "Clave": 2, "Edad": 34 }
```

Estos records se pueden unir de la siguiente forma:

```
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 de la izquierda para los casos de campos repetidos (como en el caso de *"Clave"*).

Se puede obtener el valor de cualquiera de sus campos:

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

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 se tiene un record con valores completamente distintos, también se pueden añadir al final del record original:

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

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 subrecord proyectando por alguno 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 implementación y descripción de ellas se ve más adelante.

3.2. Predicados y propiedades en Idris

Antes de ver la implementación de `consRecAuto` 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 (map fst ts))) -> Record ts ->
  Record ((l, A) :: ts)
```

Notar que se agregó el término `prf : Not (Elem l (map fst ts))`. En Idris esto representa no solo un valor `prf` de un tipo en particular, sino que representa una prueba del predicado `Not (Elem l (map fst ts))` que representa la proposición *"La etiqueta 'l' no pertenece a la lista de etiquetas de 'ts' "*. Cabe notar que como la lista de campos `ts` tiene tipo `List (String, Type)`, entonces `map fst ts` tiene tipo `List String` y representa solamente la lista de etiquetas.

Esta correspondencia entre tipos y proposiciones se conoce como isomorfismo de *Curry-Howard* [13].

El predicado `Elem` puede definirse inductivamente como cualquier otro tipo de datos, declarando sus constructores:

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

El término `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. El caso inductivo ocurre cuando se busca el elemento en el resto de la lista.

La construcción de un término del tipo `Elem x xs` constituye entonces una prueba de que `x` pertenece a la lista `xs`. Por ejemplo:

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

En el tipo de `consRec`, se tenía el tipo `Not (Elem l ts)`. `Not` es una función entre tipos que toma un tipo y retorna otro, representando la negación de un predicado. Tener un valor `prf : Not (Elem l ts)` significa que es imposible obtener una prueba de `Elem l ts`. La definición de `Not` es la siguiente:

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

`Not` es una simple función, que toma un tipo y retorna el tipo `Void`, que es un tipo sin ninguna prueba, o sea, un tipo al que no se le puede construir ningún valor ya que no tiene constructores. En lógica constructiva, representa la proposición *False*, tal que si se puede obtener una prueba de dicha proposición, entonces se puede obtener una prueba de cualquier otra. 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 un valor de 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 que la ejecución del programa llegue

a ese caso, por lo que se prueba `Void` y luego se aplica `absurd` para convencer al compilador.

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 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).

Un ejemplo simple es el siguiente:

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

En este ejemplo, si se tiene un valor de tipo `Elem x []`, y si el mismo se hubiera construido con `Here`, entonces debería tener un tipo que unifique con `Elem x (x :: xs)`, algo que es imposible ya que la lista proporcionada es vacía. De la misma forma, si se hubiera construido con `There`, entonces debería tener un tipo que unifique con `Elem x (y :: xs)`, lo cual tiene el mismo problema. Por lo tanto es imposible construir un valor del tipo `Elem x []`, por lo que si se tiene tal valor, al aplicarle la función `noEmptyElem` se puede probar Void.

En Idris, para generar una prueba de `Void`, basta con incluir un solo constructor (como `Here` en este caso).

Con la definición de `Not` vista anteriormente, se puede reescribir esta función de la siguiente forma:

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

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

Volviendo al caso de records extensibles, en las siguientes secciones se muestra la definición completa de la extensión de records. Sin embargo, primero mostraremos algunas funciones, tipos, nomenclatura y conceptos necesarios para poder hacerlo.

En primer lugar, describiremos qué son los tipos decidibles y para qué son utilizados en este trabajo.

3.2.1. Tipos decidibles

Para poder definir los records extensibles en Idris, es necesario trabajar con tipos o proposiciones decidibles. En Idris la decidibilidad de un tipo se representa de la siguiente forma:

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

Una proposición es decidible si es posible construir una prueba de ella o su negación. Si se tiene `Dec P`, entonces significa que o bien existe un valor `s : P` o existe un valor `n : Not P`.

Tener tipos decidibles es importante cuando se tienen tipos que funcionan como predicados y se necesita saber si ese predicado se cumple o no. Teniendo una prueba de `Dec P` se puede entonces realizar un análisis de casos, uno cuando `P` es verdadero y otro cuando no.

Usando la decidibilidad de la igualdad es posible testear la igualdad de valores:

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

En idris `interface` permite definir una abstracción igual a las typeclasses de Haskell, donde se parametriza por un tipo y luego se pueden construir instancias de esa interfaz/typeclass para tipos particulares.

La función `decEq t` indica que, siempre que se tienen dos elementos `x2, 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.

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 (en los ejemplos de este trabajo principalmente se trabaja con strings). Su uso es bastante simple, en vez de tener `List (String, Type)` se tiene `DecEq lty =>List (lty, Type)`. Con esta definición es posible utilizar etiquetas de otros tipos, como enumerados, naturales, etc.

3.2.2. Funciones y tipos útiles

Otras funciones y tipos útiles para este trabajo son los siguientes:

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

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

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

La función `labelsOf` toma una lista de campos y retorna la lista de sus etiquetas. Por ejemplo, `labelsOf [("Clave", Nat), ("Edad", Nat)] = ["Clave", "Edad"]`.

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

`ElemLabel` es una abreviación para el tipo de la prueba de que una etiqueta pertenece a una lista de campos.

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

La función `isElemLabel` es una función de decisión, que dada una etiqueta y una lista de campos retorna si tal etiqueta pertenece o no a esa lista. 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 del tipo de la extensión de un record:

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

Donde `ts : LabelList lty` es la lista de campos del record actual, `t` es el tipo del nuevo campo, `l` es la nueva etiqueta, `val` es el valor del nuevo campo, `Record ts` es el tipo del record a extender, `Not (ElemLabel l ts)` la prueba de que `l` no se encuentra repetida en `ts`, y el record resultante va a tener el tipo `Record ((l, t) :: ts)` con el nuevo campo.

3.2.3. 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 `ls`, se puede probar que la lista `(l, t) :: ts` no tiene etiquetas repetidas.

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` indica que la lista `ls` no tiene elementos repetidos.

Las pruebas de este predicado se contruyen de forma inductiva. Primero se prueba que la lista vacía no contiene repetidos. Luego, para el caso inductivo, si se agrega un elemento a una lista que no tiene repetidos, 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, se define el siguiente tipo:

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

De esta forma, `consRec` pasa a tener el siguiente tipo:

```
consRec : {ts : LabelList lty} ->
  {t : Type} -> (l : lty) -> (val : t) ->
  Record ts -> {isSet : IsLabelSet ((l, t) :: ts)} ->
  Record ((l, t) :: ts)
```

3.2.4. Construcción de términos de prueba

El principal problema que ocurre al utilizar la función `consRec` definida anteriormente se ubica en el parámetro `isSet : IsLabelSet ((l, t) :: ts)`. En efecto, para poder llamar a `consRec` es necesario construir una prueba de que la nueva etiqueta a agregar al record no esté repetida en el record ya existente.

Una primera opción es generar la prueba manualmente, pero esto resulta tedioso e impráctico, ya que sería necesario construir ese término de prueba cada vez que se quisiera insertar un elemento.

Una segunda opción es utilizar la decidibilidad del predicado a instanciar. Al tener un tipo decidible, es posible utilizar el typechecker para forzar la unificación del tipo decidible con si mismo o su negación. Básicamente, si uno puede generar

un valor del tipo `Dec p`, entonces puede unificarlo con `p` o `Not p` en tiempo de compilación.

En este caso en particular, sería necesario poder obtener un valor del tipo `Dec (IsLabelSet ((l, t) :: ts))`.

Para forzar la unificación, 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 _) = ()
```

La función `getYes` retorna una computación la cual hace *pattern matching* sobre el valor de tipo `Dec p`. Esta computación es ejecutada en tiempo de compilación, y tiene dos opciones: o retorna el tipo unit `()`, o retorna el tipo `p`.

En tiempo de typechecking se realiza pattern matching sobre `Dec p`. Si la prueba es de la forma `No`, entonces se retorna el valor `()` (que efectivamente es un valor del tipo `()`). Sin embargo, si la prueba es de la forma `Yes`, entonces ya se tiene un valor de tipo `p`, el cual se retorna. El pattern matching no solo permite tener una bifurcación sobre cuál valor retornar, sino también sobre cuál es el tipo de este valor retornado, pudiendo retornar dos valores con tipos totalmente distintos.

La función `getNo` es idéntica a `getYes`, pero retorna un valor de tipo `Not p` en vez de `p`.

A continuación se muestran ejemplos del uso de estas funciones:

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

En el caso de `okYes`, la función `isElem` es computada en tiempo de compilación, 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 de tipo `Elem "L1" ["L1"]`.

```
-- No compila
badYes : Elem "L1" ["L2"]
badYes = getYes $ isElem "L1" ["L2"]
```

No ocurre lo mismo para el caso de `badYes`, donde en tiempo de compilación 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 compilación.

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

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

-- No compila
badNo : Not (Elem "L1" ["L1"])
badNo = getNo $ isElem "L1" ["L1"]
```

Con este mecanismo se puede generar una prueba automática de cualquier predicado decidible, por lo que se puede simplificar el uso de `consRec`. Sin embargo,

para ello necesitamos probar que `IsLabelSet` es un tipo decidible, y necesitamos tener una función `isLabelSet` que permita obtener un valor de tipo `Dec (IsLabelSet ts)`. Para ello primero se define la función de decidibilidad para `IsSet`. Esta función, llamada `isSet`, 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
```

La función `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 sí los hay. La implementación de `isSet` realiza un análisis de casos sobre el largo de la lista. Para el caso de la lista vacía ésta no tiene elementos repetidos por definición de `isSet`. 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 recursión), y luego verificando que la cabeza de la lista no pertenezca a la cola de ésta.

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

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

La expresión dentro del `with` es dividida en sus constructores correspondientes realizando pattern matching, de forma similar a `case`. La diferencia con `case` es que `with` permite redefinir los parámetros anteriores según el resultado del matcheo de la expresión. Al ser Idris un lenguaje con tipos dependientes, pueden ocurrir situaciones donde una expresión matchea solamente cuando otros valores previos de la definición tienen valores específicos. Por ejemplo:

```
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
```

La función `eq n m` retorna true si `n` y `m` son naturales idénticos. Utiliza el costructor `Refl`, el cual es proporcionado por el lenguaje y tiene la siguiente definición:

```
data (=) : (a : Type) -> (b : Type) -> Type where
  Refl : x = x
```

En el caso de `eq`, 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 igual a `m`, se puede unificar `val` con `Refl : n = n` y cambiar `eq n m` por `eq n n` en la definición de la función.

Las funciones `ifNotSetHereThenNeitherThere` y `ifIsElemThenConsIsNotSet` son dos lemas necesarios para poder definir `isSet`.

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

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

La función `isLabelSet` simplemente permite aplicar la función `isSet` al tipo `IsLabelSet`.

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

Teniendo la función `isLabelSet` es posible generar la prueba para ser utilizada en `consRec`. Un ejemplo sería el siguiente:

```
extendedRec : Record [("Nombre", String)]
extendedRec = consRec "Nombre" "Juan"
  {isSet=(getYes $ isLabelSet ["Nombre"])}
  emptyRec
```

Con la función `getYes` se puede reescribir `consRec` para que obtenga la prueba de `IsLabelSet` de forma automática:

```
consRecAuto : DecEq lty => {ts : LabelList lty} -> {t : Type} ->
  (l : lty) -> (val : t) -> Record ts ->
  Record ((l, t) :: ts)
consRecAuto l val rec = consRec l val rec
  {isSet = getYes $ isLabelSet ((l, t) :: ts)}
```

Con esta definición ya no es necesario pasar la prueba de `IsLabelSet ((l, t) :: ts)` de forma manual.

Cabe notar que, al momento de compilar esta función, ésta resulta menos eficiente que si se hubiera utilizado una prueba de `Not (Elem l (map fst ts))` como se había definido en `consRec` originalmente. Esto se debe a que verificar si `Not (Elem l (map fst ts))` se cumple o no solamente necesita recorrer la lista `ts` una vez para buscar la etiqueta `l`. Sin embargo, para verificar si `IsLabelSet ((l, t) :: ts)` es verdad o no necesita recorrer la lista `ts`, y para cada elemento de ésta debe verificar que no pertenezca al resto de la lista. Por este motivo la aplicación de `getYes` a `isLabelSet` es cuadrática, mientras que si se aplica a `isElemLabel` ésta es lineal.

3.2.5. Generación automática de pruebas

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

A continuación se muestra una técnica alternativa que utiliza el mismo concepto del anterior, donde se realiza pattern matching sobre un tipo decidible en tiempo de compilación para unificar tipos. Sin embargo, el pattern matching se realiza en el

tipo mismo y no en una función auxiliar. Esta es una técnica estándar utilizada en lenguajes con tipos dependientes cuando se quiere trabajar con predicados decidibles.

Esto es posible gracias a este tipo y esta función:

```
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 contradicción de `p`, entonces se obtiene el tipo `()`

Este método funciona cuando se necesita unificar un tipo `type` y `TypeOrUnit dec type`. Si se tiene una prueba de `p` entonces la unificación va a dar correcta, pero si no se tiene una prueba entonces va a fallar en tiempo de compilación.

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

Como ejemplo, se tiene una función `addNat` que debe agregar un natural a una lista solamente si ya pertenece a ésta, y en caso contrario no debe compilar.

```
addNat : (n : Nat) -> (ns : List Nat) ->
  TypeOrUnit (isElem n ns) (List Nat)
addNat n ns = mkTypeOrUnit (isElem n ns)
  (\isElem => n :: ns)
```

La función `addNat` hace uso de `TypeOrUnit`, forzando a que se cumpla el predicado `Elem n ns`. Si no se cumple tal predicado, la función retorna `()`.

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

La llamada a `addNat 10 [10]` retorna el tipo `TypeOrUnit (isElem 10 [10]) (List Nat)`, pero `myListOk` espera `List Nat`. En tiempo de compilación 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 código.

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

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

```
myListBad2 : Nat -> List Nat
myListBad2 n = addNat n [10] -- 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 compilación. Por lo tanto no se puede evaluar `TypeOrUnit (isElem n [10]) (List Nat)`, lo cual hace que falle la unificación con `List Nat`.

Como conclusión, con este método es posible forzar, en tiempo de compilación, a que se cumpla un predicado específico. Si la función decidible puede ser evaluada hasta su forma normal y retorna `Yes`, entonces el código compila correctamente. Si la función decidible no puede ser evaluada hasta su forma normal, o puede ser evaluada pero retorna `No`, entonces el código no compila.

Para poder aplicar este método a los records, es necesario utilizar la función `isLabelSet` definida en la sección anterior 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 ésta sí tiene repetidos.

Con este tipo es posible tener la siguiente función 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 función es idéntica a `consRec`, solamente que no es necesario pasar una prueba de `IsLabelSet ((l, t) :: ts)`. Ahora esta prueba se calcula automáticamente en tiempo de compilación y se impacta en el tipo resultante `RecordOrUnit ((l, t) :: ts)`. Esta implementación se explica a fondo más adelante, solo basta saber que `MkRecord` es el constructor del record.

Su uso fue demostrado al comienzo de este capítulo, 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 sí se repitan, se mostrará el siguiente 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", Nat), ("A", Nat)]
      (Type of consRecAuto "A" 10
```

```
        (consRecAuto "A" 10 emptyRec))
  and
    Record [("A", Nat), ("A", Nat)] (Expected type)
```

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

3.3. Definición de un record

La implementación del tipo `Record` de este trabajo es la siguiente:

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

Esta definición se corresponde a la de HList 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`.

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

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

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 extendida con el nuevo elemento.

```
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) ::hs))
  (\isLabelSet => MkRecord isLabelSet (val :: hs))
```

3.3.1. HList actualizado

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

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

```
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. La etiqueta se guarda en forma explícita en el tipo.

Un ejemplo sería el siguiente:

```
[1, "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 cómo se implementan.

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 la misma técnica de `consRecAuto`. Las etiquetas que se pasan en la lista no tienen que estar repetidas, por lo que tal propiedad debe ser verificada en tiempo de compilación.

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

Tales etiquetas pueden o no estar en el record; si no están en el record simplemente se ignoran. El tipo de `hProjectByLabelsAuto` 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 (projectLeft ls ts)`, donde `projectLeft` es una función a nivel de tipos que 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 que se puede usar sobre listas, pero puede ser también utilizada para tener distintos tipos de un record. La computación simplemente se translada a los tipos.

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

La implementación de `projectLeft` es la siguiente:

```
projectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty
projectLeft ls [] = []
projectLeft ls ((l, ty) :: ts) with (isElem l ls)
  projectLeft ls ((l, ty) :: ts) | Yes _ =
    (l,ty) :: projectLeft ls ts
  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 y aplica el mismo razonamiento al resto de la lista. Si no pertence, no retorna ese campo y aplica la función 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} ls rec lsIsSet)
```

Simplemente aplica la misma técnica de `consRecAuto` con `TypeOrUnit`, delegando la llamada a `hProjectByLabels`, cuya implementación es la siguiente:

```
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. Comencemos por:

```
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)))
```

Donde la sintaxis `(x : a ** (P a))` indica que se tiene un par dependiente, donde el tipo del segundo valor depende del primero. Si se tiene un tipo `A` y otro tipo `B`, se puede tener un par `(A, B)`. Sin embargo, si `B` depende de un valor de `A`, se puede tener un par dependiente con `(a : A ** (B a))`.

La función anterior realiza la proyección a nivel de listas heterogéneas. También contiene otro mecanismo de manejo de tipos, donde 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
```

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

Su definición es la siguiente:

```
data IsProjectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty -> Type where
  IPL_ProjLabelElem : DecEq lty => {l : lty} ->
    Elem l ls -> IsProjectLeft {lty} ls 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
```

Como se ve, la definición de `IsProjectLeft` es muy similar a la de `projectLeft`. Esto es necesario para que se ambos sean isomorfismos (descrito por la propiedad anterior). Se pueden ver tales similitudes comparando ambas implementaciones línea a línea:

```
IPL_EmptyVect : DecEq lty => IsProjectLeft {lty} ls [] []
projectLeft ls [] []

IPL_ProjLabelElem : DecEq lty => {l : lty} ->
    Elem l ls -> IsProjectLeft {lty} ls ts res1 ->
    IsProjectLeft ls ((l,ty) :: ts) ((l,ty) :: res1)
projectLeft ls ((l, ty) :: ts) | Yes _ =
  (l,ty) :: projectLeft ls ts

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 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 sobre sus constructores que sobre los argumentos de la función original.

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

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
```

Donde el 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 no hay etiquetas repetidas en los campos originales entonces tampoco los van a haber en los campos resultantes. Su implementación se puede encontrar en el apéndice.

```
resIsProjComp = fromIsProjectLeftToComp prfLeftRes lsIsSet

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

Donde la función representa la propiedad que debe cumplir `IsProjectLeft` en relación a `projectLeft` que vimos anteriormente. Se le agrega el hecho de que `ls` no debe tener elementos 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:

```
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`. La función `sym` simplemente cambia el orden de una igualdad, por lo tanto `(sym resIsProjComp) : projectLeft ls ts1 = ts2`. Consecuentemente con esta técnica de reescritura se puede obtener un término del 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 del capítulo 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 definición es inductiva, 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 definición es la de obtener el valor de un campo de una lista heterogénea:

```
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 función toma una etiqueta de tipo `lty`, una lista heterogénea de tipo `HList ts`, y una prueba de que esa etiqueta pertenece a la lista de tipo `HasField l ts ty`. Con tales datos es posible entonces retornar el valor de tipo `ty` asociado a tal etiqueta. Su implementación realiza pattern matching sobre el predicado, obteniendo el valor `val : ty` si la etiqueta está en la cabeza de la lista, o sino lo obtiene realizando un llamado recursivo.

Con esta función se puede definir `hLookupByLabel` de la siguiente forma:

```
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 un record se define en términos de una lista heterogénea, entonces esta función simplemente aplica la función definida anteriormente a tal lista heterogénea contenida en el record.

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

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

Con esta función se podría definir la función de cálculo automático 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 es que no es posible definir `hasField`. Para implementarla, se tiene una variable `ty : Type` que puede pertenecer a 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 podría tener `hasField "Edad" [("Edad", Nat)] Nat` o `hasField "Edad" [("Edad", Nat)] String`. El primer caso debería retornar `Yes` con una prueba, mientras que el segundo deberia retornar `No` con una contradicción, ya que el tipo `Nat` y `String` son distintos. Sin embargo, no es posible realizar un chequeo `Nat = Nat` o `Not (Nat = String)`, ya que eso requeriría 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 cualesquiera en un lenguaje de tipos dependientes, verificar si son iguales no es decidible. Es decir, teóricamente, es imposible crear una instancia de `DecEq Type`.
- Si existiera tal forma, rompería el polimorfismo paramétrico. El polimorfismo paramétrico permite abstraerse de un tipo y trabajar con él sin saber específicamente cuál es. Esto permite deducir propiedades de una función solamente conociendo su tipo (esto se conoce como propiedad de parametricidad o free theorems [26]). Por ejemplo, si se tiene la función `id : a ->a`, se puede deducir que la única posible implementación es `id val = val`, ya que es imposible que la función pueda conocer información sobre el tipo `a : Type`.

  Sin embargo, si existiera una instancia de `DecEq Type`, entonces la siguiente función sería válida:

  ```
  id2 : a -> a
  ```

```
id2 {a} val with (decEq a Nat)
  id2 {a=Nat} val | Yes _ = 10
  id2 {a} val | No _ = val
```

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

A pesar de no poder utilizar `TypeOrUnit`, es posible definir una función que calcule el predicado de forma automática, 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
```

La expresión `auto hasField : HasField l ts ty` indica que el typechecker de Idris va a intentar generar una prueba del predicado `HasField l ts ty` de forma automática. Idris se da cuenta de que la definición 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. Este mecanismo se lo denominia *táctica*.

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. Próximamente, Idris intenta construir `HasFieldThere HasFieldHere`. Idris consigue en ese caso unificar el tipo de ese valor con el esperado y entonces proporciona el valor `HasFieldThere HasFieldHere` a la función `hLookupByLabelAuto`.

Como conclusión, utilizar `TypeOrUnit` es muy útil para definir estas funciones con un cálculo automático de los predicados correspondientes, pero solo puede usarse si el predicado no depende de un valor `a : Type` (o de cualquier valor que no puede verificarse por igualdad). Si es así, se deben utilizar funcionalidades propias del lenguaje, como `auto`.

3.4.3. Unión por izquierda

Al comienzo del capítulo se vió el siguiente ejemplo de unión por izquierda de records:

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

La función `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, con prioridad por los campos del record de la izquierda en caso de etiquetas repetidas.

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
```

Al obtener un record nuevo se debe computar la lista de sus campos. Esto se hace con la función `hLeftUnion_List`:

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

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

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 _ = ts
  deleteLabelAt l1 ((l2, ty) :: ts) | No _ =
    (l2, ty) :: deleteLabelAt l1 ts
```

La función `hLeftUnion_List` es un poco más compleja que las anteriores. Se define como la unión de dos listas tomando la primera, y agregándole todos los campos de la segunda, menos aquellos que tienen etiquetas que aparecen en la primera lista. Esta eliminación se realiza con `deleteLabels`, que simplemente recorre toda la lista de campos a eliminar y los elimina de la segunda (utilizando la función `deleteLabelAt`).

Al igual que en `hProjectByLabels`, `hLeftUnion` hace uso de un predicado que representa la computación de la unión de los campos del record, y delega la implementación a una función que retorna 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 esta propiedad:

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

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

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

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)
```

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

3.5. Ejemplos

En esta sección se muestra cómo se escriben los ejemplos del capítulo 2 con la solución de records extensibles de este trabajo.

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 (sea utilizando typeclasses, tipos de datos, etc).

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 has First
    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 este capítulo (como `hLeftUnion`, `hProjectLabels`, etc), 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 hasta ahora fue en algunos ejemplos básicos. En este capítulo se muestra un caso de estudio más elaborado en el cual se utilizan los records extensibles desarrollados en este trabajo para poder solucionarlo.

4.1. Descripción del caso de estudio

El caso de estudio consiste en un pequeño lenguaje de expresiones aritméticas con variables y constantes. Se define este lenguaje en Idris como un DSL (*Domain Specific Language*), lo cual permite crear términos de este lenguaje, y luego evaluarlos en Idris.

Este lenguaje está compuesto por:

- Literales dados por números naturales.
- Variables.
- Sumas de expresiones.
- Expresiones let.

Este lenguaje permite definir expresiones aritméticas formadas por valores numéricos, variables y sumas de 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)
```

La gramática del lenguaje en BNF se muestra en la Figura 4.1.

e ::= n | x | e1 + e2 | let x = n in e
Donde n ∈ Nat; x ∈ Var; e, e1 y e2 ∈ Exp

Figura 4.1: Gramática BNF

Vamos a usar records extensibles para poder que las expresiones están bien formadas, y que las llamadas al evaluador son válidas, todo en tiempo de compilación.

Dada una expresión, como `x + 3`, para evaluarla es necesario tener un ambiente con valores para sus variables libres. En el caso de `x + 3` 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 el ambiente utilizado para evaluar una expresión contiene valores para variables libres.

Los records extensibles son utilizados para 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. El ambiente puede ser extendido con nuevas variables, o se le pueden eliminar variables fuera de alcance. Al ser un record extensible se tiene la garantía de que ninguna variable se encuentra repetida en el ambiente, y por otra parte se tiene una prueba de que toda variable libre pertenece al ambiente.

Las propiedades descritas anteriormente son posibles de asegurar utilizando records extensibles y tipos dependientes. Sin estas herramientas, es posible llamar al evaluador con un ambiente sin las variables libres necesarias para evaluar la expresión, fallando 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

Toda expresión tiene un conjunto de variables libres, que se definen con las reglas de la Figura 4.2.

$$\text{lit}\ \frac{n : Nat}{n \rightsquigarrow \emptyset}$$

$$\text{var}\ \frac{v : Var}{v \rightsquigarrow \{v\}}$$

$$\text{add}\ \frac{e1 \rightsquigarrow \Delta 1 \quad e2 \rightsquigarrow \Delta 2}{e1 + e2 \rightsquigarrow \Delta 1 \cup \Delta 2}$$

$$\text{let}\ \frac{e \rightsquigarrow \Delta \quad n : Nat \quad v : Var}{\textbf{let}\ v = n\ \textbf{in}\ e \rightsquigarrow \Delta \setminus \{v\}}$$

Figura 4.2: Reglas de variables libres

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

El juicio de tipado `e : Exp fv` indica que `e` es una expresión y cuyo su conjunto de variables libres es `fv`. Como ejemplo, `e : Exp ["x", "y"]` contiene a `"x"` y `"y"` como variables libres. Se decidió incluir la información de variables libres en el tipo para facilitar la implementación del evaluador. Vamos a tomar a `String` como siendo *Var*, el dominio de los nombres de las variables.

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

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

```
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
```

Cada constructor del tipo `Exp` representa una regla de formación de expresiones. A su vez, cada constructor sigue las reglas de construcción de variables libres definidas anteriormente.

En el caso de una expresión dada por una variable la expresión contiene una única variable libre. Si es un literal entonces no hay variables libres.

En el caso de la suma de dos expresiones el conjunto de variables libres es la unión por izquierda de las listas de variables libres de los sumandos. Esto se realiza utilizando el predicado `IsLeftUnion` visto en el capítulo anterior, solo que éste funciona sobre listas `List lty` en vez de listas de campos con etiquetas `LabelList lty`.

```
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)

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 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)
```

Un ejemplo sería el siguiente:

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

En el caso de un let las variables libres corresponden a las variables libres de la expresión del let menos la variable local. Esto se realiza con el predicado `DeleteLabelAtPred_List`, que se comporta igual que `DeleteLabelAtPred` definido en el capítulo anterior, solo que se aplica sobre listas `List lty` en vez de listas de campos con etiquetas `LabelList lty`. Su definición se mostró más arriba.

Un ejemplo de tal predicado 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 (habitualmente denominados "smart constructors") 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 aplican el constructor correspondiente. Las funciones `add` y `eLet` tienen la particularidad de que realizan las computaciones sobre las listas de variables, y construyen los predicados correspondientes dadas esas computaciones. Las funciones `leftUnion` y `deleteAtList` computan valores análogos a `IsLeftUnion_List` y `DeleteAtLabel_List`, cumpliendo las siguientes propiedades:

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

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

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 usando los "smart constructors":

```
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
```

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

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

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

A diferencia de let, `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})
```

La función `local` utiliza la misma técnica que utilizamos para `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 variable varias veces, como `local [”x” := 1, ”x” := 2]`.

Tal declaración de variables se define en `LocalVariables`, que es un tipo que representa 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:

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

La implementación de `localPred` se puede ver en el apéndice, 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] $ lit 1 <-> eLet ("x" := 10) $ lit 1

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

Como `local` verifica que las variables definidas no se repitan, si se llama con variables repetidas se muestra un error de compilación:

```
e : Exp []
e = local ["x" := 10, "x" := 11] $ var "x"

  When checking right hand side of e with expected type
    Exp []

  Type mismatch between
    TypeOrUnit (isSet ["x", "x"])
      (Exp (deleteList ["x", "x"] ["x"]))
      (type of
        local ["x" := 10, "x" := 11 (var "x"))

  and
    Exp [] (Expected type)

  Specifically:
    Type mismatch between
      TypeOrUnit (isSet ["x", "x"])
        (Exp (deleteList ["x", "x"] ["x"]))
    and
      Exp []
```

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 debe tener definidos valores para todas las variables libres de la expresión.

Por ejemplo, si se tiene `Exp ["x", "y"]`, entonces deben tener ambientes que incluyan esas variables, como `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 inductivo, 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:

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

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

`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", Nats)]
```

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

Un ejemplo (con una pseudo-sintaxis) sería:

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

La evaluación de una expresión cerrada (o sea, sin variables libres) se hace de la siguiente manera:

```
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
```

Como la función `interp` solo se puede llamar con expresiones cerradas, si se llama con una expresión abierta se muestra un error de compilación:

```
v : Nat
v = interp $ var "x"

  When checking right hand side of v with expected type
    Nat

  When checking an application of function CasoDeEstudio.interp:
    Type mismatch between
      Exp [l] (Type of var l)
    and
      Exp [] (Expected type)

    Specifically:
      Type mismatch between
        [l]
      and
        []
```

La implementación completa del evaluador se muestra a continuación.

4.3.1. Literales

El caso de un literal es el más sencillo:

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

Significa retornar tal valor como un resultado.

4.3.2. Variables

Una expresión que declara una variable 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 función 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 función se utiliza 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 es posible obtener su valor. En ningún momento el lookup va a fallar en tiempo de ejecución porque no pudo encontrar la variable.

4.3.3. Sumas

Una expresión del tipo suma es sencilla de evaluar. Se necesita evaluar sus subexpresiones, 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 de tipo `Ambiente fvsEnv`, es necesario tener una prueba de `IsSubSet fvs1 fvsEnv` y `IsSubSet fvs2 fvsEnv` para poder llamar al evaluador.

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
```

Donde `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 let

Una expresión let es más difícil de evaluar que los demás casos, ya que requiere la manipulación del ambiente de variables. La implementación completa de este caso se puede ver en la Figura 4.3.

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 tenga un valor `n2` asignado en el ambiente, pero si es así, 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 al mismo al momento de evaluar la subexpresión.

Veamos cómo se realiza esto en la implementación misma:

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

```
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 recEvn 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
```

Figura 4.3: Evaluación de lets

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 fvsEnv)` 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, extendiendolo 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)
```

La función `consRec` utilizada aquí no hace uso de `IsLabelSet` sino que utiliza `Not (Elemlabel l 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 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 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 propiedad 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 fvsEnv)` 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 recordd 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`. Por eso es necesaria una prueba de `IsSubSet fvsInner fvsEnv`.

```
newSubSet = ifConsIsElemThenIsSubSet consSubSet varInEvn
```

Como se muestra más 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` es subconjunto de `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 la 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 fue el de 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). Dicho objetivo se cumplió satisfactoriamente. Es posible tener una solución al problema de records extensibles en un lenguaje como Idris, el cual presenta muchas ventajas sobre otras soluciones.

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

5.1. Viabilidad de implementar records extensibles con tipos dependientes

Este trabajo es una prueba de la viabilidad de implementar records extensibles con tipos dependientes. Una vez que se tienen claros 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:

```
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 (biblioteca de Haskell) a Idris. En este aspecto, la traducción final resultó muy similar a la de HList, y no se encontraron problemas mayores en traducir conceptos, tipos y funciones de un lenguaje a otro. El diseño actual, basado en HList, permite tener una solución adecuada al problema de records extensibles que cumple con todos los requisitos y propiedades de éste.

También se cumplió el objetivo de hacer la biblioteca amigable para el usuario final. 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 (como que no contenga etiquetas repetidas) pero sin presentarle dificultades al usuario.

Un aspecto que no resulta muy amigable al usuario es la manipulación de records no concretos. Es decir, si el usuario tiene un record concreto, como `{ "x" : 10, "y" : 15 }`, puede manipularlo de forma sencilla, ya que puede hacer uso de la generación de pruebas automáticas, como `consRecAuto`. 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)`, entre otras. A diferencia de otros lenguajes y soluciones de records extensibles, esto le agrega un nivel de complejidad mayor al uso de ellos en Idris, ya que el usuario necesita estar familiarizado con esas definiciones de propiedades y debe estar familiarizado con Idris para saber cómo generar sus pruebas y manipularlas.

En términos generales, se vió que esta solución tiene ventajas sobre otras realizadas en otros lenguajes. Comparándola con HList de Haskell, esta solución tiene el mismo poder de expresión (al ser una traducción casi directa), pero al tener tipos dependientes *first-class* tiene ventajas sobre Haskell, como por ejemplo, poder definir funciones sobre records y listas heterogéneas con los mismos mecanismos que se definen las demás funciones (y no estar forzado a utilizar typeclasses, type families, etc). Comparándola con otros lenguajes como Elm y Purescript, como se vió en el capítulo 3, esta solución permite modelar las mismas funcionalidades que en ellos, pero a su vez le da una mayor cantidad de funcionalidades y flexibilidad al usuario.

5.2. Desarrollo en Idris con tipos dependientes

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

El primer aspecto importante que se notó, es que el trabajo mayor de desarrollo no se encontraba en las definiciones de tipos y funciones 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 encuentran en el apéndice), pero varios de esos teoremas, para poder ser probados, necesitan de varios sublemas adicionales, 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 crear varios teoremas que relacionaran este predicado con los otros utilizados en la biblioteca, como:

```
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 éste 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 y mecanismos internos, 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 de uso de sus predicados y funciones, todo con el afán de facilitarle su uso. 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`. Son predicados con tipos distintos, sin embargo sus definiciones son idénticas. 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 (por ejemplo 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 los argumentos implícitos que utilizan 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 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 esos 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 continuación se describen tales formas, indicando por qué no se optó por cada una de 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 información de ellos en su tipo. Cada valor es simplemente reconocido como `HValue`, y no es posible conocer su tipo u operar con él de ninguna forma. Solo es posible insertar elementos, y manipularlos en la lista (eliminarlos, reordenarlos, etc).

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

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

Un ejemplo de su uso sería:

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

5.3.2. Existencial

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

Este enfoque se asemeja al uso de *tipos existenciales* utilizados en Haskell. Básicamente, el tipo `HList` se define como un tipo simple sin parámetros, pero sus constructores permiten utilizar valores de cualquier tipo.

Esta definición es muy similar a la que utiliza tipos dinámicos visto en la sección anterior, y tiene las mismas desventajas. Luego de creada una HList de esta forma, no es posible obtener ninguna información de los tipos de los valores que contiene, por lo que es imposible poder trabajar con tales valores. Por ese motivo tampoco fue utilizado.

Un ejemplo de su uso sería:

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

Esta HList es parametrizada por un constructor de tipos. Es decir, toma como parámetro una función que toma un tipo y construye otro tipo a partir de éste. Esta definición permite imponer una estructura en común a todos los elementos de la lista, forzando a que cada uno de ellos sea 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` que viene en la biblioteca base de Idris) permite utilizar tipos arbitrarios y obtener información de ellos accediendo a la lista de tipos, por lo cual son más útiles. Por ejemplo, con esta HList no sería posible obtener un valor de tipo `Nat`, ya que no se tiene conocimiento de que existe el tipo `Nat` en la lista heterogénea.

Algunos ejemplos son los siguientes:

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

Como se ve en este ejemplo, esta HList permite agregar etiquetas a cada uno de sus valores. Sin embargo, no se puede obtener los tipos de éstos, ya que en el tipo solo se tiene un valor `x` arbitrario, y no se puede saber cuál es su tipo (si es `Nat`, `String`, etc).

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

En este ejemplo se puede reconstruir la definición de HList existencial simplemente utilizando `HList id`. Esto indica que esta definición de HList es más poderosa que la existencial.

Capítulo 6

Trabajo a futuro

Existen posibles alternativas para mejorar la implementación de records extensibles:

- Una posible tarea a futuro es terminar de implementar las demás funciones y predicados de *HList* de Haskell que no se tradujeron a Idris en esta solución.
- En el diseño de la biblioteca y del caso de estudio, hubo varios predicados y funciones duplicados para `LabelList lty` y `List lty`. Predicados como `IsLeftUnion`, `DeleteLabelAt`, y muchos otros. Una mejora es crear predicados y funciones parametrizables por `List a`, con la restricción de que este `a` contenga una etiqueta o valor `String` o `DecEq lty`. De este modo no se repite código ni se realiza más esfuerzo en probar todos los teoremas para cada predicado en particular.
- Todavía 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). Un trabajo a futuro es investigar estos nuevos teoremas y funciones y exponerlos en la biblioteca.
- Para algunos predicados se crearon funciones que computan su mismo resultado. Por ejemplo, para el predicado `IsLeftUnion` se implementó la función `hLeftUnion_List`. 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. Alternativamente, se puede investigar para solo utilizar las funciones que computan el resultado y no utilizar los predicados (aunque esto requeriría un rediseño grande de la solución).
- 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 estas 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 pueden mejorar los mensajes de error en la generación automática de predicados. 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 para cada aplicación de `TypeOrUnit`, como por ejemplo el mensaje *"El elemento a agregar ya existe"* al llamar a `consRecAuto`.

- En el caso de estudio, se decidió definir `Exp` parametrizando el tipo por el conjunto de variables libres. Esto complejiza la definición de sus constructores, ya que en cada constructor se necesita realizar la computación del conjunto de variables libres final, cuando en sí mismo la definición de una expresión es sencilla. A su vez acopla las variables libres a la expresión, lo cual no es necesario al momento de construir expresiones, ya que el conjunto de variables libres no impone ninguna restricción sobre la construcción de tales expresiones. Una mejora es definir las expresiones con un tipo de datos simple `data Exp : Type`, y tener un predicado `data Fv : Exp ->List String ->Type` o una función `fv : Exp ->List String` para manipular las variables libres. De esta forma el tipo del evualuador sería el siguiente:

```
interpEnv : Ambiente fvsEnv -> (e : Exp) ->
  IsSubSet (fv e) fvsEnv -> Nat
```

- En el caso de estudio, las expresiones solo representan números naturales. Esto permite que el caso de estudio pueda ser resuelto sin utilizar records extensibles, sino utilizando listas heterogéneas, manteniendo en el ambiente una lista de variables libres y un valor natural para cada una. Una mejora de este caso de estudio es extenderlo con expresiones booleanas y de otros tipos. Por ejemplo, extenderlo con un caso `if-then-else` que tome una expresión booleana y dos expresiones de tipo arbitrario, y retorne una de las dos según a qué evalúa la expresión booleana. Si el lenguaje de expresiones tuviera tal extensión, entonces no sería posible resolverlo con listas heterogéneas comunes, pero sí con records extensibles. De esta forma se mostrarían los beneficios de utilizar records extensibles con una mayor claridad.

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.

[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.

[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.

[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. NOTTCS-TR-96-3. Department of Computer Science, University of Nottingham, 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.

[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». En: *Proceedings of the 1999 Haskell Workshop*. 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.

[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)*. Tallinn, 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.

[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).

[26] Philip Wadler. «Theorems for Free!» En: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, págs. 347-359.

Apéndice A

Código fuente

A continuación se muestra el código fuente de este trabajo. Primero se muestra el código fuente completo de la implementación de records extensibles descrito en el capítulo 3. Luego, se muestra el código fuente de la implementación del caso de estudio descrito en el capítulo 4.

A.1. Código fuente de records extensibles

```
module Extensible_Records

import Data.List

%default total

%access public export

symNot : Not (x = y) -> Not (y = x)
symNot notEqual Refl = notEqual Refl

consNotEqNil : {xs : List t} -> Not (x :: xs = [])
consNotEqNil Refl impossible

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

notElemInCons : Not (Elem x (y :: ys)) -> Not (Elem x ys)
notElemInCons notElemCons elemTail = notElemCons $ There elemTail

ifNotElemThenNotEqual : Not (Elem x (y :: ys)) -> Not (x = y)
ifNotElemThenNotEqual notElemCons equal =
  notElemCons $ rewrite equal in Here


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

```
ifSetThenNotElemFirst : IsSet (x :: xs) -> Not (Elem x xs)
ifSetThenNotElemFirst (IsSetCons notXIsInXs  _) = notXIsInXs

ifSetThenRestIsSet : IsSet (x :: xs) -> IsSet xs
ifSetThenRestIsSet (IsSetCons _ xsIsSet) = xsIsSet

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

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

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


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

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

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

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

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

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

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

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

recToHList : Record ts -> HList ts
```

```
recToHList (MkRecord _ hs) = hs

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

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

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

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

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

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

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

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

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


data IsProjectLeft : DecEq lty => List lty -> LabelList lty ->
  LabelList lty -> Type where
  IPL_EmptyVect : DecEq lty => IsProjectLeft {lty} ls [] []
  IPL_ProjLabelElem : DecEq lty => {l : lty} -> Elem l ls ->
    IsProjectLeft {lty} ls ts res1 ->
    IsProjectLeft ls ((l,ty) :: ts) ((l,ty) :: res1)
  IPL_ProjLabelNotElem : DecEq lty => {l : 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_EmptyVect : DecEq lty => IsProjectRight {lty} ls [] []
```

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

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

notElem_Lemma1 : Not (Elem x (y :: xs)) ->
  (Not (Elem x xs), Not (x = y))
notElem_Lemma1 notElemCons = (notElem_prf, notEq_prf)
  where
    notElem_prf isElem = notElemCons $ There isElem
    notEq_prf isEq = notElemCons $ rewrite isEq in Here

notElem_Lemma2 : Not (Elem x xs) -> Not (x = y) ->
  Not (Elem x (y :: xs))
notElem_Lemma2 notElem notEq Here = notEq Refl
```

```
notElem_Lemma2 notElem notEq (There isElem) = notElem isElem

hProjectByLabelsRightIsSet_Lemma1 : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> IsProjectRight ls ts1 ts2 ->
  Not (ElemLabel l ts1) -> Not (ElemLabel l ts2)
hProjectByLabelsRightIsSet_Lemma1 IPR_EmptyVect notElem = notElem
hProjectByLabelsRightIsSet_Lemma1
  (IPR_ProjLabelElem isElem subPrjRight) notElem =
  let
    (notElemSub, notEq) = notElem_Lemma1 notElem
    notIsElemRec =
      hProjectByLabelsRightIsSet_Lemma1 subPrjRight notElemSub
  in notIsElemRec
hProjectByLabelsRightIsSet_Lemma1
  (IPR_ProjLabelNotElem subNotElem subPrjRight) notElem =
  let
    (notElemSub, notEq) = notElem_Lemma1 notElem
    notIsElemRec =
      hProjectByLabelsRightIsSet_Lemma1 subPrjRight notElemSub
  in notElem_Lemma2 notIsElemRec notEq

hProjectByLabelsLeftIsSet_Lemma1 : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> IsProjectLeft ls ts1 ts2 ->
  Not (ElemLabel l ts1) -> Not (ElemLabel l ts2)
hProjectByLabelsLeftIsSet_Lemma1 IPL_EmptyVect notElem = notElem
hProjectByLabelsLeftIsSet_Lemma1
  (IPL_ProjLabelElem isElem subPrjLeft) notElem =
  let
    (notElemSub, notEq) = notElem_Lemma1 notElem
    notIsElemRec =
      hProjectByLabelsLeftIsSet_Lemma1 subPrjLeft notElemSub
  in notElem_Lemma2 notIsElemRec notEq
hProjectByLabelsLeftIsSet_Lemma1
  (IPL_ProjLabelNotElem subNotElem subPrjLeft) notElem =
  let
    (notElemSub, notEq) = notElem_Lemma1 notElem
    notIsElemRec =
      hProjectByLabelsLeftIsSet_Lemma1 subPrjLeft notElemSub
  in notIsElemRec

hProjectByLabelsRightIsSet_Lemma2 : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> IsProjectRight ls ts1 ts2 ->
  IsLabelSet ts1 -> IsLabelSet ts2
hProjectByLabelsRightIsSet_Lemma2 IPR_EmptyVect isLabelSet =
  isLabelSet
hProjectByLabelsRightIsSet_Lemma2
  (IPR_ProjLabelElem isElem subPrjRight)
  (IsSetCons notMember subLabelSet) =
  let isLabelSetRec =
    hProjectByLabelsRightIsSet_Lemma2 subPrjRight subLabelSet
  in isLabelSetRec
hProjectByLabelsRightIsSet_Lemma2
  (IPR_ProjLabelNotElem notElem subPrjRight)
  (IsSetCons notMember subLabelSet) =
```

```
  let isLabelSetRec =
        hProjectByLabelsRightIsSet_Lemma2 subPrjRight subLabelSet
      notElemPrf =
        hProjectByLabelsRightIsSet_Lemma1 subPrjRight notMember
  in IsSetCons notElemPrf isLabelSetRec

hProjectByLabelsLeftIsSet_Lemma2 : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> IsProjectLeft ls ts1 ts2 ->
  IsLabelSet ts1 -> IsLabelSet ts2
hProjectByLabelsLeftIsSet_Lemma2 IPL_EmptyVect isLabelSet =
  isLabelSet
hProjectByLabelsLeftIsSet_Lemma2
  (IPL_ProjLabelElem isElem subPrjLeft)
  (IsSetCons notMember subLabelSet) =
  let isLabelSetRec =
        hProjectByLabelsLeftIsSet_Lemma2 subPrjLeft subLabelSet
      notElemPrf =
        hProjectByLabelsLeftIsSet_Lemma1 subPrjLeft notMember
  in IsSetCons notElemPrf isLabelSetRec
hProjectByLabelsLeftIsSet_Lemma2
  (IPL_ProjLabelNotElem notElem subPrjLeft)
  (IsSetCons notMember subLabelSet) =
  let isLabelSetRec =
        hProjectByLabelsLeftIsSet_Lemma2 subPrjLeft subLabelSet
  in isLabelSetRec

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

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

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

```
fromIsProjectLeftToComp_Lemma_1 : DecEq lty => {ls : List lty} ->
  [] = projectLeft ls []
fromIsProjectLeftToComp_Lemma_1 {ls=[]} = Refl
fromIsProjectLeftToComp_Lemma_1 {ls=(l::ls)} = Refl

fromIsProjectLeftToComp_Lemma_2 : DecEq lty => {ls : List lty} ->
  {ts : LabelList lty} -> Not (Elem l ls) ->
  projectLeft ls ts = projectLeft ls ((l, ty) :: ts)
fromIsProjectLeftToComp_Lemma_2 {l} {ls} {ts} notLInLs
  with (isElem l ls)
  fromIsProjectLeftToComp_Lemma_2 {l} {ls} {ts} notLInLs |
    Yes lInLs = absurd $ notLInLs lInLs
  fromIsProjectLeftToComp_Lemma_2 {l} {ls} {ts} notLInLs |
    No _  = Refl

fromIsProjectLeftToComp : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> IsProjectLeft ls ts1 ts2 ->
  IsSet ls -> ts2 = projectLeft ls ts1
fromIsProjectLeftToComp {ls} {ts1=[]} {ts2=[]} IPL_EmptyVect _ =
  fromIsProjectLeftToComp_Lemma_1 {ls=ls}
fromIsProjectLeftToComp {ls} {ts1=(l1,ty1) :: ts1}
  (IPL_ProjLabelElem l1InLs isProjLeft) lsIsSet with (isElem l1 ls)
  fromIsProjectLeftToComp {ls} {ts1=(l1,ty1) :: ts1}
    (IPL_ProjLabelElem l1InLs isProjLeft) lsIsSet | Yes _ =
    cong $ fromIsProjectLeftToComp isProjLeft lsIsSet
  fromIsProjectLeftToComp {ls} {ts1=(l1,ty1) :: ts1}
    (IPL_ProjLabelElem l1InLs isProjLeft) lsIsSet | No notL1InLs =
    absurd $ notL1InLs l1InLs
fromIsProjectLeftToComp {ls} {ts1=(l1,ty1) :: ts1}
  (IPL_ProjLabelNotElem notIsElem isProjLeft) lsIsSet =
  let subPrf = fromIsProjectLeftToComp isProjLeft lsIsSet
      resEq = fromIsProjectLeftToComp_Lemma_2 notIsElem {ls=ls}
        {ts=ts1} {l=l1} {ty=ty1}
  in rewrite subPrf in (rewrite resEq in Refl)

fromCompToIsProjectLeft : DecEq lty => (ls : List lty) ->
  (ts : LabelList lty) -> IsProjectLeft ls ts (projectLeft ls ts)
fromCompToIsProjectLeft ls [] = IPL_EmptyVect
fromCompToIsProjectLeft ls ((l,ty) :: ts) with (isElem l ls)
  fromCompToIsProjectLeft ls ((l,ty) :: ts) | Yes lInLs =
    let subPrf = fromCompToIsProjectLeft ls ts
    in IPL_ProjLabelElem lInLs subPrf
  fromCompToIsProjectLeft ls ((l,ty) :: ts) | No notLInLs =
    let subPrf = fromCompToIsProjectLeft ls ts
    in IPL_ProjLabelNotElem notLInLs subPrf

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

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)

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 => Not (l1 = l2) ->
    DeleteLabelAtPred {lty} l1 ts1 ts2 ->
    DeleteLabelAtPred l1 ((l2,ty) :: ts1) ((l2,ty) :: ts2)

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

fromDeleteLabelAtFuncToPred : DecEq lty => {l : lty} ->
  {ts : LabelList lty} ->
  DeleteLabelAtPred l ts (deleteLabelAt l ts)
fromDeleteLabelAtFuncToPred {l} {ts=[]} = EmptyRecord
fromDeleteLabelAtFuncToPred {l=l1} {ts=((l2,ty) :: ts)}
  with (decEq l1 l2)
  fromDeleteLabelAtFuncToPred {l=l1} {ts=((l1,ty) :: ts)} |
    Yes Refl = IsElem
  fromDeleteLabelAtFuncToPred {l=l1} {ts=((l2,ty) :: ts)} |
    No notL1EqL2 =
    let subDelPred = fromDeleteLabelAtFuncToPred {l=l1} {ts}
    in IsNotElem notL1EqL2 subDelPred

fromDeleteLabelAtPredToFunc : DecEq lty => {l : lty} ->
  {ts1, ts2 : LabelList lty} ->
  DeleteLabelAtPred l ts1 ts2 -> ts2 = deleteLabelAt l ts1
fromDeleteLabelAtPredToFunc {ts1=[]} EmptyRecord = Refl
fromDeleteLabelAtPredToFunc {l=l2} {ts1 = (l2, ty) :: ts2}
  {ts2 = ts2} IsElem with (decEq l2 l2)
  fromDeleteLabelAtPredToFunc {l=l2} {ts1 = (l2, ty) :: ts2}
    {ts2 = ts2} IsElem | Yes Refl = Refl
  fromDeleteLabelAtPredToFunc {l=l2} {ts1 = (l2, ty) :: ts2}
    {ts2 = ts2} IsElem | No notEq = absurd $ notEq Refl
```

```
fromDeleteLabelAtPredToFunc {l=l1} {ts1 = (l2, ty) :: ts1}
    {ts2 = ((l2, ty) :: ts2)} (IsNotElem notElem subDel)
    with (decEq l1 l2)
  fromDeleteLabelAtPredToFunc {l=l1} {ts1 = (l1, ty) :: ts1}
    {ts2 = ((l1, ty) :: ts2)} (IsNotElem notElem subDel) |
    Yes Refl =
    absurd $ notElem Refl
  fromDeleteLabelAtPredToFunc {l=l1} {ts1 = (l2, ty) :: ts1}
    {ts2 = ((l2, ty) :: ts2)} (IsNotElem notElem subDel) |
    No notL1EqL2 =
    let subPrf = fromDeleteLabelAtPredToFunc subDel
    in cong subPrf

fromIsProjectRightToDeleteLabelAtPred : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  IsProjectRight [l] ts1 ts2 -> DeleteLabelAtPred l ts1 ts2
fromIsProjectRightToDeleteLabelAtPred IPR_EmptyVect = EmptyRecord
fromIsProjectRightToDeleteLabelAtPred
  (IPR_ProjLabelElem isElem IPR_EmptyVect) impossible
fromIsProjectRightToDeleteLabelAtPred
  (IPR_ProjLabelElem isElem
    (IPR_ProjLabelElem subElem subProjRight)) impossible
fromIsProjectRightToDeleteLabelAtPred
  (IPR_ProjLabelElem isElem
    (IPR_ProjLabelNotElem subNotElem subProjRight)) impossible
fromIsProjectRightToDeleteLabelAtPred
  (IPR_ProjLabelNotElem notElem subPrjRight) =
  let subDelFromRec =
        fromIsProjectRightToDeleteLabelAtPred subPrjRight
      notEqual = ifNotElemThenNotEqual notElem
  in IsNotElem (symNot notEqual) subDelFromRec


hDeleteAtLabelHList : DecEq lty => {ts1 : LabelList lty} ->
  (l : lty) -> HList ts1 ->
  (ts2 : LabelList lty ** (HList ts2, DeleteLabelAtPred l ts1 ts2))
hDeleteAtLabelHList l hs =
  let (_, (ts2 ** (hs2, prjRightRes))) = hProjectByLabelsHList [l] hs
  in (ts2 **
    (hs2, fromIsProjectRightToDeleteLabelAtPred prjRightRes))

hDeleteAtLabelIsNotElem : DecEq lty => {ts1, ts2 : LabelList lty} ->
  {l1, l2 : lty} -> DeleteLabelAtPred l1 ts1 ts2 ->
  Not (ElemLabel l2 ts1) -> Not (ElemLabel l2 ts2)
hDeleteAtLabelIsNotElem EmptyRecord notL2InTs1 l2InTs2 =
  noEmptyElem l2InTs2
hDeleteAtLabelIsNotElem IsElem notL2InTs1 l2InTs2 =
  notElemInCons notL2InTs1 l2InTs2
hDeleteAtLabelIsNotElem {l1} {l2}
  (IsNotElem {l2} {ty} notL1EqL3 delAtPred) notL2InTs1 Here =
  ifNotElemThenNotEqual notL2InTs1 Refl
hDeleteAtLabelIsNotElem {l1} {l2}
  (IsNotElem {l2=l3} {ty} notL1EqL3 delAtPred) notL2InTs1Cons
  (There l2InTs2) =
```

```
  let notL2InTs1 = notElemInCons notL2InTs1Cons
      notL2InTs2  = hDeleteAtLabelIsNotElem delAtPred notL2InTs1
  in notL2InTs2 l2InTs2

hDeleteAtLabelIsLabelSet : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  DeleteLabelAtPred l ts1 ts2 -> IsLabelSet ts1 -> IsLabelSet ts2
hDeleteAtLabelIsLabelSet EmptyRecord _ = IsSetNil
hDeleteAtLabelIsLabelSet IsElem (IsSetCons notLInTs1 isSetTs1) =
  isSetTs1
hDeleteAtLabelIsLabelSet {l=l1}
  (IsNotElem {l2} {ty} notL1EqL2 delAtPred)
  (IsSetCons notL2InTs1 isSetTs1) =
  let isSetTs2 = hDeleteAtLabelIsLabelSet delAtPred isSetTs1
      notL2InTs2 = hDeleteAtLabelIsNotElem delAtPred notL2InTs1
  in IsSetCons notL2InTs2 isSetTs2

hDeleteAtLabel : DecEq lty => {ts1 : LabelList lty} ->
  (l : lty) -> Record ts1 ->
  (ts2 : LabelList lty ** (Record ts2, DeleteLabelAtPred l ts1 ts2))
hDeleteAtLabel l rec =
  let
    isSetTs1 = recLblIsSet rec
    hs = recToHList rec
    (ts2 ** (hs2, delAtPred)) = hDeleteAtLabelHList l hs
    isSetTs2 = hDeleteAtLabelIsLabelSet delAtPred isSetTs1
  in (ts2 ** (hListToRec {prf=isSetTs2} hs2, delAtPred))

(++) : HList ts1 -> HList ts2 -> HList (ts1 ++ ts2)
(++) [] hs2 = hs2
(++) (h1 :: hs1) hs2 = h1 :: (hs1 ++ hs2)

ifIsElemThenIsInAppendLeft : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} -> ElemLabel l ts1 ->
  ElemLabel l (ts1 ++ ts2)
ifIsElemThenIsInAppendLeft {ts1=((l,ty) :: ts1)} Here = Here
ifIsElemThenIsInAppendLeft {ts1=((l,ty) :: ts1)} {ts2}
  (There isThere) =
  let subPrf = ifIsElemThenIsInAppendLeft {ts2=ts2} isThere
  in (There subPrf)

ifIsElemThenIsInAppendRight : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  ElemLabel l ts2 -> ElemLabel l (ts1 ++ ts2)
ifIsElemThenIsInAppendRight {ts1=[]} isElem = isElem
ifIsElemThenIsInAppendRight {ts1=((l1,ty1) :: ts1)} {ts2=[]}
  isElem = absurd $ noEmptyElem isElem
ifIsElemThenIsInAppendRight {l} {ts1=((l1,ty1) :: ts1)}
  {ts2=((l2,ty2) :: ts2)} isElem =
  let subPrf = ifIsElemThenIsInAppendRight {ts1=ts1}
        {ts2=((l2,ty2)::ts2)} {l=l} isElem
  in There subPrf

ifIsInOneThenIsInAppend : DecEq lty =>
```

```
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  Either (ElemLabel l ts1) (ElemLabel l ts2) ->
  ElemLabel l (ts1 ++ ts2)
ifIsInOneThenIsInAppend (Left isElem) =
  ifIsElemThenIsInAppendLeft isElem
ifIsInOneThenIsInAppend {ts1} {ts2} {l} (Right isElem) =
  ifIsElemThenIsInAppendRight isElem

ifIsInAppendThenIsInOne : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  ElemLabel l (ts1 ++ ts2) ->
  Either (ElemLabel l ts1) (ElemLabel l ts2)
ifIsInAppendThenIsInOne {ts1=[]} isElem = (Right isElem)
ifIsInAppendThenIsInOne {ts1=((l1,ty1) :: ts1)} Here = (Left Here)
ifIsInAppendThenIsInOne {l} {ts1=((l1,ty) :: ts1)} (There isThere) =
  case (ifIsInAppendThenIsInOne isThere) of
    Left isElem => Left $ There isElem
    Right isElem => Right isElem

ifNotInAppendThenNotInNeither : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  Not (ElemLabel l (ts1 ++ ts2)) ->
  (Not (ElemLabel l ts1), Not (ElemLabel l ts2))
ifNotInAppendThenNotInNeither {ts1=[]} {ts2=ts2} {l} notInAppend =
  (notInTs1, notInTs2)
  where
    notInTs1 : Not (ElemLabel l [])
    notInTs1 inTs1 = noEmptyElem inTs1

    notInTs2 : Not (ElemLabel l ts2)
    notInTs2 inTs2 = notInAppend inTs2
ifNotInAppendThenNotInNeither {ts1=((l2,ty) :: ts1)} {ts2} {l}
  notInAppend = (notInTs1, notInTs2)
  where
    notInTs1 : Not (ElemLabel l ((l2,ty)::ts1))
    notInTs1 Here impossible
    notInTs1 (There isThere) =
      let isInAppend = ifIsInOneThenIsInAppend {ts1=ts1}
            {ts2=ts2} (Left isThere)
      in notInAppend (There isInAppend)

    notInTs2 : Not (ElemLabel l ts2)
    notInTs2 inTs2 =
      let isInAppend = ifIsInOneThenIsInAppend {ts1=ts1}
            {ts2=ts2} (Right inTs2)
      in notInAppend (There isInAppend)

ifNotInEitherThenNotInAppend : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} ->
  Not (ElemLabel l ts1) -> Not (ElemLabel l ts2) ->
  Not (ElemLabel l (ts1 ++ ts2))
ifNotInEitherThenNotInAppend {ts1=[]} notInTs1 notInTs2 inAppend =
  notInTs2 inAppend
ifNotInEitherThenNotInAppend {ts1=((l1,ty1) :: ts1)} notInTs1
```

```
  notInTs2 Here = notInTs1 Here
ifNotInEitherThenNotInAppend {ts1=((l1,ty1) :: ts1)} notInTs1
  notInTs2 (There inThere) =
  let notInAppend = ifNotInEitherThenNotInAppend
        (notElemInCons notInTs1) notInTs2
  in notInAppend inThere

ifAppendIsSetThenEachIsToo : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> IsLabelSet (ts1 ++ ts2) ->
  (IsLabelSet ts1, IsLabelSet ts2)
ifAppendIsSetThenEachIsToo {ts1=[]} isSet = (IsSetNil, isSet)
ifAppendIsSetThenEachIsToo {ts1=((l,ty) :: ts1)}
  (IsSetCons notInAppend isSetAppend) =
  let
    subPrf = ifAppendIsSetThenEachIsToo isSetAppend
    notInTs1 = fst $ ifNotInAppendThenNotInNeither notInAppend
  in (IsSetCons notInTs1 (fst $ subPrf), snd subPrf)

hAppend : DecEq lty => {ts1, ts2 : LabelList lty} -> Record ts1 ->
  Record ts2 -> IsLabelSet (ts1 ++ ts2) -> Record (ts1 ++ ts2)
hAppend rec1 rec2 isLabelSet =
  let
    hs1 = recToHList rec1
    hs2 = recToHList rec2
  in hListToRec {prf=isLabelSet} (hs1 ++ hs2)

hAppendAuto : DecEq lty => {ts1, ts2 : LabelList lty} ->
  Record ts1 -> Record ts2 -> RecordOrUnit (ts1 ++ ts2)
hAppendAuto {ts1} {ts2} rec1 rec2 =
  mkTypeOrUnit (isLabelSet (ts1 ++ ts2))
    (\isSet => hAppend rec1 rec2 isSet)

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

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

fromDeleteLabelsFuncToPred : DecEq lty => {ls : List lty} ->
  {ts : LabelList lty} ->
  DeleteLabelsPred ls ts (deleteLabels ls ts)
fromDeleteLabelsFuncToPred {ls=[]} {ts} = EmptyLabelList
fromDeleteLabelsFuncToPred {ls=l :: ls} {ts} =
  let subDelLabelPred = fromDeleteLabelsFuncToPred {ls} {ts}
      delLabelAtPred = fromDeleteLabelAtFuncToPred {l}
```

```
        {ts=deleteLabels ls ts}
  in DeleteFirstOfLabelList {tsAux=deleteLabels ls ts}
       delLabelAtPred subDelLabelPred

fromDeleteLabelsPredToFunc : DecEq lty => {ls : List lty} ->
  {ts1, ts2 : LabelList lty} -> DeleteLabelsPred ls ts1 ts2 ->
  ts2 = deleteLabels ls ts1
fromDeleteLabelsPredToFunc {ls=[]} EmptyLabelList = Refl
fromDeleteLabelsPredToFunc {ls=l :: ls}
  (DeleteFirstOfLabelList delAt subDelLabels) =
  let subPrfDelAt = fromDeleteLabelAtPredToFunc delAt
      subPrfDelLabels = fromDeleteLabelsPredToFunc subDelLabels
  in rewrite (sym subPrfDelLabels) in subPrfDelAt

hDeleteLabelsHList : DecEq lty => {ts1 : LabelList lty} ->
  (ls : List lty) -> HList ts1 ->
  (ts2 : LabelList lty ** (HList ts2, DeleteLabelsPred ls ts1 ts2))
hDeleteLabelsHList {ts1} [] hs = (ts1 ** (hs, EmptyLabelList))
hDeleteLabelsHList (l :: ls) hs1 =
  let (ts3 ** (hs2, delLabelPred)) = hDeleteLabelsHList ls hs1
      (ts4 ** (hs3, delAtPred)) = hDeleteAtLabelHList l hs2
  in (ts4 ** (hs3, DeleteFirstOfLabelList delAtPred delLabelPred))

hDeleteLabelsIsLabelSet : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {ls : List lty} ->
  DeleteLabelsPred ls ts1 ts2 ->
  IsLabelSet ts1 -> IsLabelSet ts2
hDeleteLabelsIsLabelSet EmptyLabelList isSetTs1 = isSetTs1
hDeleteLabelsIsLabelSet (DeleteFirstOfLabelList {tsAux=ts3}
  delAtLabel delLabels) isSetTs1 =
  let isSetTs3 = hDeleteLabelsIsLabelSet delLabels isSetTs1
      isSetTs2 = hDeleteAtLabelIsLabelSet delAtLabel isSetTs3
  in isSetTs2

hDeleteLabels : DecEq lty => {ts1 : LabelList lty} ->
  (ls : List lty) -> Record ts1 ->
  (ts2 : LabelList lty ** (Record ts2, DeleteLabelsPred ls ts1 ts2))
hDeleteLabels ls rec =
  let
    isSetTs1 = recLblIsSet rec
    hs = recToHList rec
    (ts2 ** (hs2, delLabelsPred)) = hDeleteLabelsHList ls hs
    isSetTs2 = hDeleteLabelsIsLabelSet delLabelsPred isSetTs1
  in
    (ts2 ** (hListToRec {prf=isSetTs2} hs2, delLabelsPred))

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)

hLeftUnion_List : DecEq lty => LabelList lty -> LabelList lty ->
```

```
  LabelList lty
hLeftUnion_List ts1 ts2 = ts1 ++ (deleteLabels (labelsOf ts1) ts2)

fromHLeftUnionFuncToPred : DecEq lty =>
  {ts1, ts2 : LabelList lty} ->
  IsLeftUnion ts1 ts2 (hLeftUnion_List ts1 ts2)
fromHLeftUnionFuncToPred {ts1} {ts2} =
  let delLabelsPred =
        fromDeleteLabelsFuncToPred {ls=labelsOf ts1} {ts=ts2}
  in IsLeftUnionAppend delLabelsPred

fromHLeftUnionPredToFunc : DecEq lty =>
  {ts1, ts2, ts3 : LabelList lty} -> IsLeftUnion ts1 ts2 ts3 ->
  ts3 = hLeftUnion_List ts1 ts2
fromHLeftUnionPredToFunc (IsLeftUnionAppend delLabels) =
  let eqDelLabels = fromDeleteLabelsPredToFunc delLabels
  in cong eqDelLabels

ifDeleteLabelsThenAppendIsSetLemma_1_1 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {t : (lty,Type)} ->
  IsLabelSet (ts1 ++ (t :: ts2)) -> IsLabelSet (ts1 ++ ts2)
ifDeleteLabelsThenAppendIsSetLemma_1_1 {ts1=[]}
  (IsSetCons notElem isSet) = isSet
ifDeleteLabelsThenAppendIsSetLemma_1_1 {ts1=((l,ty) :: ts1)}
  (IsSetCons notElem isSet) =
  let subPrf = ifDeleteLabelsThenAppendIsSetLemma_1_1 isSet
      (notInTs1, notInTs2Cons) =
        ifNotInAppendThenNotInNeither notElem
      notInTs2 = notElemInCons notInTs2Cons
      notInAppend = ifNotInEitherThenNotInAppend notInTs1 notInTs2
  in IsSetCons notInAppend subPrf

ifDeleteLabelsThenAppendIsSetLemma_1_2_1 : DecEq lty =>
  {l1, l2 : lty} -> {ts1, ts2 : LabelList lty} ->
  Not (ElemLabel l1 (ts1 ++ ((l2, ty2) :: ts2))) ->
  Not (ElemLabel l2 ts1) -> Not (Elem l2 (l1 :: labelsOf (ts1)))
ifDeleteLabelsThenAppendIsSetLemma_1_2_1 {ts1} {ts2} {l1} {l2=l1}
  {ty2} notElemCons notL2InTs1 Here =
  let inCons = ifIsInOneThenIsInAppend {l=l1} {ts1}
        {ts2=(l1,ty2) :: ts2} (Right Here)
  in notElemCons inCons
ifDeleteLabelsThenAppendIsSetLemma_1_2_1 notElemCons notL2InTs1
  (There isThere) = notL2InTs1 isThere

ifDeleteLabelsThenAppendIsSetLemma_1_2 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} -> {ty : Type} ->
  IsLabelSet (ts1 ++ ((l,ty) :: ts2)) ->
  (Not (ElemLabel l ts1), Not (ElemLabel l ts2))
ifDeleteLabelsThenAppendIsSetLemma_1_2 {ts1=[]} {l} {ty} {ts2}
  (IsSetCons notElem subIsSet) = (noEmptyElem, notElem)
ifDeleteLabelsThenAppendIsSetLemma_1_2 {ts1=(l1,ty1) :: ts1}
  {l=l2} {ty=ty2} {ts2} (IsSetCons notElem subIsSet) =
  let (notInTs1, notInTs2) =
        ifDeleteLabelsThenAppendIsSetLemma_1_2 subIsSet
```

```
  in (ifDeleteLabelsThenAppendIsSetLemma_1_2_1 notElem notInTs1,
       notInTs2)

ifDeleteLabelsThenAppendIsSetLemma_1_3 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l1, l2 : lty} ->
  DeleteLabelAtPred l1 ts1 ts2 -> Not (ElemLabel l2 ts1) ->
  Not (ElemLabel l2 ts2)
ifDeleteLabelsThenAppendIsSetLemma_1_3 EmptyRecord notInTs1 inTs2 =
  notInTs1 inTs2
ifDeleteLabelsThenAppendIsSetLemma_1_3 IsElem notInTs1 inTs2 =
  notInTs1 $ There inTs2
ifDeleteLabelsThenAppendIsSetLemma_1_3
  (IsNotElem notEqual subDelAt) notInTs1 Here = notInTs1 Here
ifDeleteLabelsThenAppendIsSetLemma_1_3 {l1} {l2}
  {ts1=((l3,ty3) :: ts1)} {ts2=((l3,ty3) :: ts2)}
  (IsNotElem notEqual subDelAt) notInTs1 (There inTs2) =
  let subPrf = ifDeleteLabelsThenAppendIsSetLemma_1_3 {l1=l1}
        {ts1=ts1} {ts2=ts2} subDelAt (notElemInCons notInTs1)
  in subPrf inTs2

ifDeleteLabelsThenAppendIsSetLemma_1_4_1 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l1, l2 : lty} -> {ty2 : Type} ->
  Not (l1 = l2) -> Not (ElemLabel l1 (ts1 ++ ts2)) ->
  Not (ElemLabel l1 (ts1 ++ ((l2,ty2) :: ts2)))
ifDeleteLabelsThenAppendIsSetLemma_1_4_1 {ts1=[]} notEqual
  notInAppend Here = notEqual Refl
ifDeleteLabelsThenAppendIsSetLemma_1_4_1 {ts1=[]} notEqual
  notInAppend (There isThere) = notInAppend isThere
ifDeleteLabelsThenAppendIsSetLemma_1_4_1 {l1} {l2}
  {ts1=(l1,ty3) :: ts1} notEqual notInAppend Here = notInAppend Here
ifDeleteLabelsThenAppendIsSetLemma_1_4_1 {l1} {l2}
  {ts1=(l3,ty3) :: ts1} {ts2=ts2} {ty2=ty2} notEqual notInAppend
  (There isThere) =
  let subPrf = ifDeleteLabelsThenAppendIsSetLemma_1_4_1 {l1=l1}
        {l2=l2} {ts1=ts1} {ts2=ts2} {ty2=ty2} notEqual
        (notElemInCons notInAppend)
  in subPrf isThere

ifDeleteLabelsThenAppendIsSetLemma_1_4 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} -> {ty : Type} ->
  IsLabelSet (ts1 ++ ts2) -> Not (ElemLabel l ts1) ->
  Not (ElemLabel l ts2) -> IsLabelSet (ts1 ++ ((l,ty) :: ts2))
ifDeleteLabelsThenAppendIsSetLemma_1_4 {ts1=[]} isSet notInTs1
  notInTs2 = IsSetCons notInTs2 isSet
ifDeleteLabelsThenAppendIsSetLemma_1_4 {l=l2} {ty=ty2}
  {ts1=(l1,ty1) :: ts1} {ts2} (IsSetCons notElem isSet)
  notInTs1 notInTs2 =
  let subPrf = ifDeleteLabelsThenAppendIsSetLemma_1_4 {l=l2}
        {ty=ty2} isSet (notElemInCons notInTs1) notInTs2
      notEqual = symNot $ ifNotElemThenNotEqual notInTs1
      notElemCons = ifDeleteLabelsThenAppendIsSetLemma_1_4_1
        {l2=l2} {ty2=ty2} {l1=l1} {ts1=ts1} {ts2=ts2} notEqual
        notElem
  in IsSetCons notElemCons subPrf
```

```
ifDeleteLabelsThenAppendIsSetLemma_1 : DecEq lty =>
  {ts1, ts2, ts3 : LabelList lty} -> {l : lty} ->
  DeleteLabelAtPred l ts2 ts3 -> IsLabelSet (ts1 ++ ts2) ->
  IsLabelSet (ts1 ++ ts3)
ifDeleteLabelsThenAppendIsSetLemma_1 EmptyRecord isSet = isSet
ifDeleteLabelsThenAppendIsSetLemma_1 {ts1=[]} IsElem
  (IsSetCons notElem isSet) = isSet
ifDeleteLabelsThenAppendIsSetLemma_1 {l} {ts1=((l1,ty1) :: ts1)}
  {ts3} IsElem (IsSetCons notElem isSet) =
  let (notInTs1, notInTs3Cons) =
        ifNotInAppendThenNotInNeither notElem
      notInTs3 = notElemInCons notInTs3Cons
      notInAppend = ifNotInEitherThenNotInAppend notInTs1 notInTs3
  in IsSetCons notInAppend
       (ifDeleteLabelsThenAppendIsSetLemma_1_1 {ts1=ts1}
       {ts2=ts3} isSet)
ifDeleteLabelsThenAppendIsSetLemma_1 {ts1=ts1}
  {ts2=((l2,ty2) :: ts2)} {ts3=((l2,ty2) :: ts3)}
  (IsNotElem notElem delAt) isSet =
  let isSetAppend = ifDeleteLabelsThenAppendIsSetLemma_1_1 isSet
      subPrf = ifDeleteLabelsThenAppendIsSetLemma_1 {ts1=ts1}
        {ts2=ts2} {ts3=ts3} delAt isSetAppend
      (notInTs1, notInTs2) = ifDeleteLabelsThenAppendIsSetLemma_1_2
        isSet {l=l2} {ty=ty2} {ts1=ts1} {ts2=ts2}
      notInTs3 =
        ifDeleteLabelsThenAppendIsSetLemma_1_3 delAt notInTs2
  in ifDeleteLabelsThenAppendIsSetLemma_1_4 subPrf notInTs1 notInTs3

ifDeleteLabelsThenAppendIsSetLemma_2 : DecEq lty =>
  {ts1, ts2 : LabelList lty} -> {l : lty} -> IsLabelSet ts1 ->
  DeleteLabelAtPred l ts1 ts2 -> Not (ElemLabel l ts2)
ifDeleteLabelsThenAppendIsSetLemma_2 isSet1 EmptyRecord elemLabel =
  noEmptyElem elemLabel
ifDeleteLabelsThenAppendIsSetLemma_2 (IsSetCons notElem isSet)
  IsElem elemLabel = notElem elemLabel
ifDeleteLabelsThenAppendIsSetLemma_2 (IsSetCons notElem isSet)
  (IsNotElem notEqual subDelAt) Here = notEqual Refl
ifDeleteLabelsThenAppendIsSetLemma_2 {l=l}
  (IsSetCons notElem isSet) (IsNotElem notEqual subDelAt)
  (There isThere) =
  let subPrf =
        ifDeleteLabelsThenAppendIsSetLemma_2 {l=l} isSet subDelAt
  in subPrf isThere

ifDeleteLabelsThenAppendIsSetLemma : DecEq lty =>
  {ts1, ts2, tsDel : LabelList lty} -> IsLabelSet ts1 ->
  IsLabelSet ts2 -> DeleteLabelsPred (labelsOf ts1) ts2 tsDel ->
  IsLabelSet (ts1 ++ tsDel)
ifDeleteLabelsThenAppendIsSetLemma {ts1=[]} isSet1 isSet2
  EmptyLabelList = isSet2
ifDeleteLabelsThenAppendIsSetLemma {ts1=((l1,ty1) :: ts1)} {tsDel}
  (IsSetCons notElem subIsSet1) isSet2 (DeleteFirstOfLabelList
  {tsAux=ts3} subDelAt subDel) =
```

```
  let
    subPrf = ifDeleteLabelsThenAppendIsSetLemma {ts1=ts1} subIsSet1
      isSet2 subDel
    resIsSet = ifDeleteLabelsThenAppendIsSetLemma_1 {ts1=ts1}
      {ts2=ts3} {ts3=tsDel} subDelAt subPrf
    isSet3 = snd $ ifAppendIsSetThenEachIsToo subPrf
    isNotInTsDel =
      ifDeleteLabelsThenAppendIsSetLemma_2 isSet3 subDelAt
    isNotInAppend = ifNotInEitherThenNotInAppend notElem isNotInTsDel
  in IsSetCons isNotInAppend resIsSet

hLeftUnionPred : DecEq lty => {ts1, ts2 : LabelList lty} ->
  Record ts1 -> Record ts2 ->
  (tsRes : LabelList lty **
    (Record tsRes, IsLeftUnion ts1 ts2 tsRes))
hLeftUnionPred {ts1} {ts2} rec1 rec2 =
  let
    isSet1 = recLblIsSet rec1
    isSet2 = recLblIsSet rec2
    (tsDel ** (recDel, prfDel)) = hDeleteLabels (labelsOf ts1) rec2
    recRes = hAppend rec1 recDel
      (ifDeleteLabelsThenAppendIsSetLemma {ts1=ts1} {ts2=ts2}
      {tsDel=tsDel} isSet1 isSet2 prfDel)
   in
    (ts1 ++ tsDel ** (recRes, IsLeftUnionAppend prfDel))

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

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

noEmptyHasField : Not (HasField l [] ty)
noEmptyHasField HasFieldHere impossible
noEmptyHasField (HasFieldThere _) impossible

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

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

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

hUpdateAtLabel_HList : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> ty -> HList ts -> HasField l ts ty -> HList ts
hUpdateAtLabel_HList l val1 (val2 :: hs) HasFieldHere = val1 :: hs
hUpdateAtLabel_HList l val1 (val2 :: hs)
  (HasFieldThere hasFieldThere) =
  val2 :: (hUpdateAtLabel_HList l val1 hs hasFieldThere)

hUpdateAtLabel : DecEq lty => {ts : LabelList lty} -> (l : lty) ->
  ty -> Record ts -> HasField l ts ty -> Record ts
hUpdateAtLabel {ts} l val rec hasField =
  let
    isLabelSet = recLblIsSet rec
    hs = recToHList rec
  in
    hListToRec {prf=isLabelSet}
      (hUpdateAtLabel_HList {ts=ts} l val hs hasField)

hUpdateAtLabelAuto : DecEq lty => {ts : LabelList lty} ->
  (l : lty) -> ty -> Record ts ->
  {auto hasField : HasField l ts ty} -> Record ts
hUpdateAtLabelAuto {ts} l val rec {hasField} =
  hUpdateAtLabel {ts} l val rec hasField

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

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

A.2. Código fuente del caso de estudio

```
module CasoDeEstudio

import Data.List
import Extensible_Records

%default total

%access public export

deleteAtList : DecEq lty => lty -> List lty -> List lty
deleteAtList _ [] = []
deleteAtList l1 (l2 :: ls) with (decEq l1 l2)
  deleteAtList l1 (l1 :: ls) | Yes Refl = ls
  deleteAtList l1 (l2 :: ls) | No _ = l2 :: deleteAtList l1 ls

deleteList : DecEq lty => List lty -> List lty -> List lty
deleteList [] ls = ls
```

```
deleteList (l :: ls1) ls2 =
  let subDelete = deleteList ls1 ls2
  in deleteAtList l subDelete

leftUnion : DecEq lty => List lty -> List lty -> List lty
leftUnion ls1 ls2 = ls1 ++ (deleteList ls1 ls2)

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)

fromDeleteLabelAtListFuncToPred : DecEq lty => {l : lty} ->
  {ls : List lty} ->
  DeleteLabelAtPred_List l ls (deleteAtList l ls)
fromDeleteLabelAtListFuncToPred {l} {ls = []} = EmptyRecord_List
fromDeleteLabelAtListFuncToPred {l=l1} {ls = (l2 :: ls)}
  with (decEq l1 l2)
  fromDeleteLabelAtListFuncToPred {l=l1} {ls = (l1 :: ls)} |
    Yes Refl = IsElem_List
  fromDeleteLabelAtListFuncToPred {l=l1} {ls = (l2 :: ls)} |
    No notL1EqL2 =
    let subDelPred = fromDeleteLabelAtListFuncToPred {l=l1} {ls}
    in IsNotElem_List notL1EqL2 subDelPred

fromDeleteLabelsListFuncToPred : DecEq lty =>
  {ls1, ls2 : List lty} ->
  DeleteLabelsPred_List ls1 ls2 (deleteList ls1 ls2)
fromDeleteLabelsListFuncToPred {ls1 = []} {ls2} =
  EmptyLabelList_List
fromDeleteLabelsListFuncToPred {ls1 = (l :: ls1)} {ls2} =
  let subDelListPred = fromDeleteLabelsListFuncToPred {ls1} {ls2}
      delAtPred =
        fromDeleteLabelAtListFuncToPred {l} {ls=deleteList ls1 ls2}
  in DeleteFirstOfLabelList_List {lsAux=deleteList ls1 ls2}
       delAtPred subDelListPred
```

```
fromLeftUnionFuncToPred : DecEq lty => {ls1, ls2 : List lty} ->
  IsLeftUnion_List {lty} ls1 ls2 (leftUnion ls1 ls2)
fromLeftUnionFuncToPred {ls1} {ls2} =
  let delPred = fromDeleteLabelsListFuncToPred {ls1} {ls2}
  in IsLeftUnionAppend_List delPred

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

infixr 2 :=

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

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})

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)
localPred {localVars=[]} {fvsInner} _ innerExp = innerExp
localPred {localVars=l :: localVars} (varDec :: vars) innerExp
  {isSet = (IsSetCons _ isSet)} =
  let subExp = localPred vars innerExp {isSet}
  in eLet varDec subExp

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})

data IsSubSet : List lty -> List lty -> Type where
```

```
  IsSubSetNil : IsSubSet [] ls
  IsSubSetCons : IsSubSet ls1 ls2 -> Elem l ls2 ->
    IsSubSet (l :: ls1) ls2

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

labelsOfAllNats : labelsOf (AllNats ls) = ls
labelsOfAllNats {ls = []} = Refl
labelsOfAllNats {ls = l :: ls} = cong $ labelsOfAllNats {ls}

ifNotElemThenNotInNats : Not (Elem x xs) ->
  Not (ElemLabel x (AllNats xs))
ifNotElemThenNotInNats {xs = []} notXInXs xInLabelXs =
  absurd $ noEmptyElem xInLabelXs
ifNotElemThenNotInNats {xs = x1 :: xs} notXInXs Here =
  notXInXs Here
ifNotElemThenNotInNats {xs = x1 :: xs} notXInXs (There there) =
  let notInCons = notElemInCons notXInXs
      subPrf = ifNotElemThenNotInNats notInCons
  in absurd $ subPrf there

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

ifAppendIsSubSetThenSoIsEach : DecEq lty =>
  {ls1, ls2, ls3 : List lty} -> IsSubSet (ls1 ++ ls2) ls3 ->
  (IsSubSet ls1 ls3, IsSubSet ls2 ls3)
ifAppendIsSubSetThenSoIsEach {ls1=[]} subSet =
  (IsSubSetNil, subSet)
ifAppendIsSubSetThenSoIsEach {ls1=l1::ls1} {ls2} {ls3}
  (IsSubSetCons subSet elem) =
  let (subPrfLeft, subPrfRight) =
        ifAppendIsSubSetThenSoIsEach {ls1} {ls2} {ls3} subSet
  in (IsSubSetCons subPrfLeft elem, subPrfRight)

ifIsSubSetOfEachThenIsSoAppend : DecEq lty =>
  {ls1, ls2, ls3 : List lty} -> IsSubSet ls1 ls3 ->
  IsSubSet ls2 ls3 -> IsSubSet (ls1 ++ ls2) ls3
ifIsSubSetOfEachThenIsSoAppend {ls1 = []} subSetLs1 subSetLs2 =
  subSetLs2
ifIsSubSetOfEachThenIsSoAppend {ls1 = l1 :: ls1}
  (IsSubSetCons subSetLs1 l1InLs3) subSetLs2 =
  let subPrf = ifIsSubSetOfEachThenIsSoAppend subSetLs1 subSetLs2
  in IsSubSetCons subPrf l1InLs3

ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 : DecEq lty =>
  {ls1, ls2, ls3, ls4 : List lty} -> IsSubSet (ls1 ++ ls2) ls3 ->
  Elem l ls3 -> DeleteLabelAtPred_List l ls4 ls2 ->
  IsSubSet (ls1 ++ ls4) ls3
ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 subSet elem
  EmptyRecord_List = subSet
ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 subSet elem IsElem_List =
```

```
  let (ls1SubSetLs3, ls2SubSetLs3) =
        ifAppendIsSubSetThenSoIsEach subSet
      ls2ConsSubSetLs3 = IsSubSetCons ls2SubSetLs3 elem
  in ifIsSubSetOfEachThenIsSoAppend ls1SubSetLs3 ls2ConsSubSetLs3
ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 subSet lInLs3
  (IsNotElem_List notEq delAt) =
  let (ls1SubSetLs3, ls2ConsSubSetLs3) =
        ifAppendIsSubSetThenSoIsEach subSet
      IsSubSetCons ls2SubSetLs3 l2InLs3 = ls2ConsSubSetLs3
      ls1AppLs2SubSetLs3 =
        ifIsSubSetOfEachThenIsSoAppend ls1SubSetLs3 ls2SubSetLs3
      subPrf =
        ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 ls1AppLs2SubSetLs3
          lInLs3 delAt
      (_, ls4SubSetLs3) = ifAppendIsSubSetThenSoIsEach subPrf
      ls4ConsSubSetLs3 = IsSubSetCons ls4SubSetLs3 l2InLs3
  in ifIsSubSetOfEachThenIsSoAppend ls1SubSetLs3 ls4ConsSubSetLs3

ifIsSubSetThenLeftUnionIsSubSet_Lemma_2 : DecEq lty =>
  {ls1, ls2, ls3, ls4 : List lty} -> IsSubSet (ls1 ++ ls2) ls3 ->
  DeleteLabelsPred_List ls1 ls4 ls2 -> IsSubSet ls4 ls3
ifIsSubSetThenLeftUnionIsSubSet_Lemma_2 {ls1=[]} subSet
  EmptyLabelList_List = subSet
ifIsSubSetThenLeftUnionIsSubSet_Lemma_2 {ls1=l1 :: ls1}
  (IsSubSetCons subSet elem)
  (DeleteFirstOfLabelList_List delAt delLabels) =
  let isSubSetAux =
        ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 subSet elem delAt
  in ifIsSubSetThenLeftUnionIsSubSet_Lemma_2 isSubSetAux delLabels

ifIsSubSetThenLeftUnionIsSubSet : DecEq lty =>
  {ls1, ls2, lsSub1, lsSub2 : List lty} -> IsSubSet ls1 ls2 ->
  IsLeftUnion_List lsSub1 lsSub2 ls1 ->
  (IsSubSet lsSub1 ls2, IsSubSet lsSub2 ls2)
ifIsSubSetThenLeftUnionIsSubSet subSet
  (IsLeftUnionAppend_List delLabels) =
  (isSubSetLeft subSet delLabels, isSubSetRight subSet delLabels)
  where
    isSubSetLeft :  DecEq lty =>
      {rs2, rs3, rsSub1, rsSub2 : List lty} ->
      IsSubSet (rsSub1 ++ rs3) rs2 ->
      DeleteLabelsPred_List rsSub1 rsSub2 rs3 ->
      IsSubSet rsSub1 rs2
    isSubSetLeft {rsSub1} {rs3} {rs2} subSet delLabels =
      fst $ ifAppendIsSubSetThenSoIsEach subSet

    isSubSetRight :  DecEq lty =>
      {rs2, rs3, rsSub1, rsSub2 : List lty} ->
      IsSubSet (rsSub1 ++ rs3) rs2 ->
      DeleteLabelsPred_List rsSub1 rsSub2 rs3 ->
      IsSubSet rsSub2 rs2
    isSubSetRight {rsSub1 = []} subSet2 EmptyLabelList_List =
      subSet2
    isSubSetRight {rsSub1 = r :: rsSub1}
```

```
      (IsSubSetCons subSet elem)
      (DeleteFirstOfLabelList_List delAt delLabels) =
      let auxIsSubSet =
            ifIsSubSetThenLeftUnionIsSubSet_Lemma_1 subSet elem
              delAt
      in ifIsSubSetThenLeftUnionIsSubSet_Lemma_2 auxIsSubSet
           delLabels

ifHasFieldInElemThenItHasThere : DecEq lty => {ls : List lty} ->
  Elem l ls -> HasField l (AllNats ls) Nat
ifHasFieldInElemThenItHasThere {ls = []} elem =
  absurd $ noEmptyElem elem
ifHasFieldInElemThenItHasThere {l = l2} {ls = l2 :: ls} Here =
  HasFieldHere
ifHasFieldInElemThenItHasThere {l = l1} {ls = l2 :: ls}
  (There later) =
  HasFieldThere $ ifHasFieldInElemThenItHasThere later

ifIsSubSetThenHasFieldInIt : DecEq lty =>
  {ls1, ls2 : List lty} -> IsSubSet ls1 ls2 ->
  HasField l (AllNats ls1) Nat -> HasField l (AllNats ls2) Nat
ifIsSubSetThenHasFieldInIt {ls1 = []} _ hasField =
  absurd $ noEmptyHasField hasField
ifIsSubSetThenHasFieldInIt {l=l1} {ls1 = (l2 :: ls1)} subSet
  hasField with (decEq l1 l2)
  ifIsSubSetThenHasFieldInIt {l=l1} {ls1 = (l1 :: ls1)}
  (IsSubSetCons subSet elem) hasField | Yes Refl =
    ifHasFieldInElemThenItHasThere elem
  ifIsSubSetThenHasFieldInIt {l=l1} {ls1 = (l1 :: ls1)}
    (IsSubSetCons subSet elem) HasFieldHere | No notL1EqL2 =
    absurd $ notL1EqL2 Refl
  ifIsSubSetThenHasFieldInIt {l=l1} {ls1 = (l2 :: ls1)}
    (IsSubSetCons subSet elem) (HasFieldThere hasFieldThere) |
    No notL1EqL2 =
    ifIsSubSetThenHasFieldInIt subSet hasFieldThere

ifIsSubSetThenIsSubSetOfCons : IsSubSet ls1 ls2 ->
  IsSubSet ls1 (l :: ls2)
ifIsSubSetThenIsSubSetOfCons IsSubSetNil = IsSubSetNil
ifIsSubSetThenIsSubSetOfCons {l=l1}
  (IsSubSetCons {l=l2} subSet l1InLs2) =
  let subPrf = ifIsSubSetThenIsSubSetOfCons subSet
  in IsSubSetCons subPrf (There l1InLs2)

ifIsSubSetThenIsSubSetWhenAddingElem : IsSubSet ls1 ls2 ->
  IsSubSet (l :: ls1) (l :: ls2)
ifIsSubSetThenIsSubSetWhenAddingElem subSet =
  IsSubSetCons (ifIsSubSetThenIsSubSetOfCons subSet) Here

ifConsIsElemThenIsSubSet : IsSubSet ls1 (l :: ls2) ->
  Elem l ls2 -> IsSubSet ls1 ls2
ifConsIsElemThenIsSubSet IsSubSetNil isElem = IsSubSetNil
ifConsIsElemThenIsSubSet (IsSubSetCons isSubSet Here) isElem =
  let subPrf = ifConsIsElemThenIsSubSet isSubSet isElem
```

```
  in IsSubSetCons subPrf isElem
ifConsIsElemThenIsSubSet (IsSubSetCons isSubSet (There later))
  isElem =
  let subPrf = ifConsIsElemThenIsSubSet isSubSet isElem
  in IsSubSetCons subPrf later

ifIsSubSetThenSoIfYouDeleteLabel :
  DeleteLabelAtPred_List l ls1 ls3 -> IsSubSet ls3 ls2 ->
  IsSubSet ls1 (l :: ls2)
ifIsSubSetThenSoIfYouDeleteLabel EmptyRecord_List subSet =
  IsSubSetNil
ifIsSubSetThenSoIfYouDeleteLabel IsElem_List subSet =
  ifIsSubSetThenIsSubSetWhenAddingElem subSet
ifIsSubSetThenSoIfYouDeleteLabel (IsNotElem_List notEq delAt)
  (IsSubSetCons subSet elem) =
  let subPrf = ifIsSubSetThenSoIfYouDeleteLabel delAt subSet
  in IsSubSetCons subPrf (There elem)

ifIsElemThenHasFieldNat : Elem l ls -> HasField l (AllNats ls) Nat
ifIsElemThenHasFieldNat Here = HasFieldHere
ifIsElemThenHasFieldNat (There later) =
  HasFieldThere $ ifIsElemThenHasFieldNat later

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 (Lit 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

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