Desarrollo de records extensibles en lenguajes con tipos dependientes

Gonzalo Waszczuk

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

14 de Junio, 2017

Records

Estructura de datos estática que contiene valores de (posiblemente) diferentes tipos, los cuales están asociados a etiquetas.

Records

Estructura de datos estática que contiene valores de (posiblemente) diferentes tipos, los cuales están asociados a etiquetas.

Example (Record)

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

Records Extensibles

Record que se puede extender de forma dinámica, agregando nuevos valores con sus correspondientes etiquetas.

Records Extensibles

Record que se puede extender de forma dinámica, agregando nuevos valores con sus correspondientes etiquetas.

Example (Definición de record)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan", edad = 20 }
```

Records Extensibles

Record que se puede extender de forma dinámica, agregando nuevos valores con sus correspondientes etiquetas.

Example (Definición de record)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan", edad = 20 }
```

Example (Extensión de record)

```
p' :: Record { apellido :: String, nombre :: String,
       edad :: Int }
p' = { apellido = "Torres" } .*. p
```

Records Extensibles

Record que se puede extender de forma dinámica, agregando nuevos valores con sus correspondientes etiquetas.

Example (Definición de record)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan", edad = 20 }
```

Example (Extensión de record)

```
p' :: Record { apellido :: String, nombre :: String,
       edad :: Int }
p' = { apellido = "Torres" } .*. p
```

Example (Definición alternativa)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan" } .*.
    { edad = 20 } .*.
    emptyRec
```

Otras operaciones (1)

Example (Lookup)

```
n :: String
n = p .!. nombre
-- "Juan"
```

Otras operaciones (1)

Example (Lookup)

```
n :: String
n = p .!. nombre
-- "Juan"
```

Example (Update)

```
p' :: Record { nombre :: String, edad :: Int }
p' = updateR nombre "Pedro" p
-- { nombre = "Pedro", edad = 20 }
```

Otras operaciones (2)

Example (Delete)

```
p' :: Record { edad :: Int }
p' = p .//. nombre
-- { edad = 20 }
```

Otras operaciones (2)

Example (Delete)

```
p' :: Record { edad :: Int }
p' = p .//. nombre
-- { edad = 20 }
```

Example (Union)

```
p' :: Record { nombre :: String, apellido :: String }
p' = { nombre = "Juan", apellido = "Torres" }

p'' :: Record { nombre :: String, edad :: Int,
        apellido :: String}
p'' = p .||. p''
-- { nombre = "Juan", edad = 20, apellido = "Torres" }
```

Por qué usar records extensibles?

- Estructura dinámica de datos. Se puede extender, se puede reducir.
- Valores con tipos arbitrarios y variables.
- Fuertemente tipada. Se puede conocer el tipo de cada campo en particular.
- Ejemplos de uso: Intérprete de un lenguaje, consultas a base de datos, archivo de configuracion, etc.

	Estructura homogénea	**Estructura heterogénea**
Estático		Record
Dinámico	Diccionario	Record Extensible

Cuadro: Comparación con otras estructuras de datos

Lenguajes actuales con records extensibles

Lenguajes proporcionan records extensibles como primitivas del lenguaje, o mediante bibliotecas de usuario.

Lenguajes actuales con records extensibles

Lenguajes proporcionan records extensibles como primitivas del lenguaje, o mediante bibliotecas de usuario.

Como primitiva del lenguaje

Los records extensibles son funcionalidades del lenguaje en sí, y tienen sintaxis y funcionamiento especial en el lenguaje.
Ejemplos: Elm, Purescript.

Lenguajes actuales con records extensibles

Lenguajes proporcionan records extensibles como primitivas del lenguaje, o mediante bibliotecas de usuario.

Como primitiva del lenguaje

Los records extensibles son funcionalidades del lenguaje en sí, y tienen sintaxis y funcionamiento especial en el lenguaje.
Ejemplos: Elm, Purescript.

Como biblioteca de usuario

Los records extensibles se definen como componentes dentro del lenguaje que pueden ser utilizados por usuarios mediante bibliotecas.
Ejemplos: records, rawr, vinyl, HList en Haskell.

Qué se busca en este trabajo?

Se busca tener una solución de records extensibles que:

- Sea typesafe.
- Cubra todas las funcionalidades y operaciones de records extensibles.
- Se defina como biblioteca de usuario.
- Tenga records extensibles como first-class citizens.
- Sea práctica de usar.

Qué se busca en este trabajo?

Se busca tener una solución de records extensibles que:

- Sea typesafe.
- Cubra todas las funcionalidades y operaciones de records extensibles.
- Se defina como biblioteca de usuario.
- Tenga records extensibles como first-class citizens.
- Sea práctica de usar.

La biblioteca HList de Haskell cumple con estas funcionalidades. Este trabajo se basa en HList, adaptándolo a tipos dependientes.

Idris

- Solución de este trabajo basada en Idris.
- Idris es un lenguaje funcional, total, fuertemente tipado, y con tipos dependientes.
- En un sistema de tipos con tipos dependientes, los tipos pueden depender de valores y son first-class citizens.

Records como listas heterogéneas

Example (Ejemplo original)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan" } .*.
    { edad = 20 } .*.
    emptyRec
```

Records como listas heterogéneas

Example (Ejemplo original)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan" } .*.
    { edad = 20 } .*.
    emptyRec
```

Un record extensible se puede ver como una lista variable de campos con valores. Estos campos tienen tipos distintos, por lo que el record se puede representar con una *lista heterogénea*.

Records como listas heterogéneas

Example (Ejemplo original)

```
p :: Record { nombre :: String, edad :: Int }
p = { nombre = "Juan" } .*.
    { edad = 20 } .*.
    emptyRec
```

Un record extensible se puede ver como una lista variable de campos con valores. Estos campos tienen tipos distintos, por lo que el record se puede representar con una *lista heterogénea*.

Example (Record como lista heterogénea)

```
p = { nombre = "Juan" } ::
    { edad = 20 } ::
    Nil
```

Listas heterogéneas en Idris

Definition (HList en Idris)

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

Listas heterogéneas en Idris

Definition (HList en Idris)

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

Definition (Operación sobre HList)

```
hLength : HList ts -> Nat
hLength HNil      = 0
hLength (x :> xs) = 1 + hLength xs
```

Record definido usando HList

Example

```
p : HList [String, Int]
p = "Juan" :> 20 :> HNil
```

HList con etiquetas

Donde están las etiquetas? Falta completar la definición.

HList con etiquetas

Donde están las etiquetas? Falta completar la definición.

Definition (HList extendido)

```
data LHList : List (lty, Type) -> Type where
  HNil : LHList []
  (:>) : Field lty t -> LHList ts -> LHList ((lty, t) :: ts)

data Field : lty -> Type -> Type where
  (.=.) : (l : lty) -> (v : t) -> Field l t
```

HList con etiquetas

Donde están las etiquetas? Falta completar la definición.

Definition (HList extendido)

```
data LHList : List (lty, Type) -> Type where
  HNil : LHList []
  (:>) : Field lty t -> LHList ts -> LHList ((lty, t) :: ts)

data Field : lty -> Type -> Type where
  (.=.) : (l : lty) -> (v : t) -> Field l t
```

Example (Record definido usando LHList)

```
p : LHList [("nombre", String), ("edad", Int)]
p = ("nombre" .=. "Juan) :>
    ("edad" .=. 20) :>
    HNil
```

Etiquetas repetidas

Example (Record con etiquetas repetidas)

```
p : LHList [("nombre", String), ("nombre", Int)]
p = ("nombre" .=. "Juan) :>
    ("nombre" .=. 20) :>
    HNil
```

Etiquetas repetidas

Example (Record con etiquetas repetidas)

```
p : LHList [("nombre", String), ("nombre", Int)]
p = ("nombre" .=. "Juan) :>
    ("nombre" .=. 20) :>
    HNil
```

Este caso compila, pero es incorrecto. La definición de records extensibles no debe permitir construir records con etiquetas repetidas.

Definición completa de Record

Un record extensible es una lista heterogénea de campos con etiquetas, tal que ninguna etiqueta esté repetida.

Definición completa de Record

Un record extensible es una lista heterogénea de campos con etiquetas, tal que ninguna etiqueta esté repetida.

Definition (Definición de un record extensibles)

```
data Record : List (lty, Type) -> Type where
  MkRecord : IsSet (labelsOf ts) -> LHList ts ->
                                    Record ts
```

Definition

```
labelsOf : List (lty, Type) -> List lty
labelsOf = map fst
```

IsSet

Predicado inductivo sobre listas que indica que no hay valores repetidos.

IsSet

Predicado inductivo sobre listas que indica que no hay valores repetidos.

Definition (Definición de IsSet)

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

IsSet

Predicado inductivo sobre listas que indica que no hay valores repetidos.

Definition (Definición de IsSet)

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

`Elem` es un predicado inductivo que indica que un valor pertenece a una lista.

Definition (Definición de Elem)

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

Construcción final de un record (1)

Con estas definiciones se puede construir un record:

Example (Ejemplo de record extensible)

```
r : Record [("nombre", String), ("edad", Int)]
r = MkRecord (IsSetCons not1 (IsSetCons not2 IsSetNil))
    (("nombre" .=. "Juan") :>
    ("edad" .=. 20) :>
    HNil)

not1 : Not (Elem "nombre" ["edad"])
not1 Here impossible

not2 : Not (Elem "edad" [])
not2 Here impossible
```

Construcción final de un record (2)

Example (Record vacío)

```
emptyRec : Record []
emptyRec = MkRecord IsSetNil HNil
```

Extensión de un record

La extensión de un record simplemente verifica que la etiqueta no esté repetida, y luego agrega el campo a la lista de campos del record.

Extensión de un record

La extensión de un record simplemente verifica que la etiqueta no esté repetida, y luego agrega el campo a la lista de campos del record.

Definition

```
consR : Field l t -> Not (Elem l (labelsOf ts))
                  -> Record ts
                  -> Record ((l, t) :: ts)
consR f p (MkRecord isS fs)
  = MkRecord (IsSetCons p isS) (f :> fs)
```

Extensión de un record

La extensión de un record simplemente verifica que la etiqueta no esté repetida, y luego agrega el campo a la lista de campos del record.

Definition

```
consR : Field l t -> Not (Elem l (labelsOf ts))
                  -> Record ts
                  -> Record ((l, t) :: ts)
consR f p (MkRecord isS fs)
  = MkRecord (IsSetCons p isS) (f :> fs)
```

Example (Ejemplo de record extensible)

```
r : Record [("nombre", String), ("edad", Int)]
r = consR ("nombre" .=. "Juan") not1 $
    consR ("edad" .=. 20) not2 $
    emptyRec
```

Generación de pruebas automáticas (1)

La definición anterior tiene un problema. Siempre es necesario proveer la prueba de no pertenencia.
Podemos mejorarlo, generando la prueba en tiempo de compilación de forma automática.

Generación de pruebas automáticas (2)

Para poder generar las pruebas, utilizamos el hecho que la pertenencia es un predicado decidible.
Un predicado es decidible si siempre se puede conseguir una prueba de ella o de su negación.

Generación de pruebas automáticas (2)

Para poder generar las pruebas, utilizamos el hecho que la pertenencia es un predicado decidible.
Un predicado es decidible si siempre se puede conseguir una prueba de ella o de su negación.

Definition (Definición de decidibilidad en Idris)

```
data Dec : Type -> Type where
  Yes : prop     -> Dec prop
  No  : Not prop -> Dec prop

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

Generación de pruebas automáticas (2)

Para poder generar las pruebas, utilizamos el hecho que la pertenencia es un predicado decidible.
Un predicado es decidible si siempre se puede conseguir una prueba de ella o de su negación.

Definition (Definición de decidibilidad en Idris)

```
data Dec : Type -> Type where
  Yes : prop     -> Dec prop
  No  : Not prop -> Dec prop

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

Definition (Función de decidibilidad de Elem)

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

Generación de pruebas automáticas (3)

Se puede usar la decidibilidad de un predicado para obtener una prueba de ella o de su negación en tiempo de compilación. Según el resultado, se unifica con el tipo esperado o se unifica con unit.

Generación de pruebas automáticas (3)

Se puede usar la decidibilidad de un predicado para obtener una prueba de ella o de su negación en tiempo de compilación. Según el resultado, se unifica con el tipo esperado o se unifica con unit.

Definition

```
MaybeE : DecEq lty => lty -> List (lty, Type) -> Type -> Type
MaybeE l ts r = UnitOrType (isElem l (labelsOf ts)) r

UnitOrType : Dec p -> Type -> Type
UnitOrType (Yes _) _   = ()
UnitOrType (No no) res = res

mkUorT : (d : Dec p) -> (f : Not p -> res)
                     -> UnitOrType d res
mkUorT (Yes _) _ = ()
mkUorT (No no) f = f no
```

Nueva definición de extensión de record (1)

Con esta nueva técnica, se puede redefinir consR para que genere la prueba de no inclusión de etiquetas de manera automática.

Nueva definición de extensión de record (1)

Con esta nueva técnica, se puede redefinir consR para que genere la prueba de no inclusión de etiquetas de manera automática.

Definition

```
(.*.) : DecEq lty => {l : lty} ->
        Field l t -> Record ts ->
        MaybeE l ts (Record ((l, t) :: ts))
(.*.) {ts} f@(l .=. v) (MkRecord isS fs)
  = mkUorT (isElem l (labelsOf ts))
           (\notp => MkRecord (IsSetCons notp isS) (f :> fs))
```

Nueva definición de extensión de record (2)

Example (Ejemplo original con nueva definición de extensión)

```
r : Record [("nombre", String), ("edad", Int)]
r = ("nombre" .=. "Juan") .*.
    ("edad" .=. 20) .*.
    emptyRec
```

Lookup (1)

Para implementar lookup se precisa una prueba de que el campo deseado existe y tiene el tipo que se busca.

Lookup (1)

Para implementar lookup se precisa una prueba de que el campo deseado existe y tiene el tipo que se busca.

Definition (Existencia de campo)

```
data HasField : lty -> Type -> List (lty, Type) -> Type where
  HFHere  : HasField l ty ((l, ty) :: ts)
  HFThere : HasField l ty ts ->
            HasField l ty ((l', ty') :: ts)
```

Lookup (1)

Para implementar lookup se precisa una prueba de que el campo deseado existe y tiene el tipo que se busca.

Definition (Existencia de campo)

```
data HasField : lty -> Type -> List (lty, Type) -> Type where
  HFHere  : HasField l ty ((l, ty) :: ts)
  HFThere : HasField l ty ts ->
            HasField l ty ((l', ty') :: ts)
```

Example

```
HasField "nombre" String [("nombre", String), ("edad", Int)]
HasField "edad"   Int    [("nombre", String), ("edad", Int)]
```

Lookup (2)

Lookup actúa recursivamente sobre el término de prueba para obtener el campo buscado.

Definition

```
lookupR : (l : lty) -> Record ts -> HasField l ty ts -> ty
lookupR l (MkRecord _ fs) hasF = lookupH l fs hasF

lookupH : (l : lty) -> LHList ts -> HasField l ty ts -> ty
lookupH _ ((_ .=. v) :: _) HFHere    = v
lookupH l (_ :: ts) (HFThere hasTh) = lookupH l ts hasTh
```

Lookup (3)

No se puede generar la prueba de HasField automáticamente utilizando la técnica vista anteriormente porque el predicado no es decidible.
Sin embargo, sí se puede generar automáticamente utilizando la funcionalidad 'auto' de Idris.

Lookup (3)

No se puede generar la prueba de HasField automáticamente utilizando la técnica vista anteriormente porque el predicado no es decidible.
Sin embargo, sí se puede generar automáticamente utilizando la funcionalidad 'auto' de Idris.

Definition

```
(.!.) : (l : lty) -> Record ts ->
        {auto hasF : HasField l ty ts} -> ty
l .!. r {hasF} = lookupR l r hasF
```

Lookup (3)

No se puede generar la prueba de HasField automáticamente utilizando la técnica vista anteriormente porque el predicado no es decidible.
Sin embargo, sí se puede generar automáticamente utilizando la funcionalidad 'auto' de Idris.

Definition

```
(.!.) : (l : lty) -> Record ts ->
        {auto hasF : HasField l ty ts} -> ty
l .!. r {hasF} = lookupR l r hasF
```

Example

```
nombre : String
nombre = "nombre" .!. r
-- "Juan"
```

Otras operaciones (1)

Append

Concatena los campos de dos records si no hay etiquetas repetidas.

Definition

```
(.++.) : DecEq lty => {ts : List (lty, Type)} -> Record ts ->
         Record us -> IsSet (ts ++ us) -> Record (ts ++ us)
```

Example

```
r' : Record [("apellido", String)]
r' = ("apellido" .=. "Torres") .*. empty

r'' : Record [("apellido", String), ("nombre", String),
      ("edad", Int)]
r'' = r' .++. r
-- { "apellido" = "Torres", "nombre" = "Juan", "edad" = 20 }
```

Otras operaciones (2)

Project

Permite seleccionar campos de un record a retornar según si pertenecen a una lista dada o no.

Definition

```
project : DecEq lty => {ts : List (lty, Type)} ->
          (ls : List lty) -> Record ts -> IsSet ls ->
          (Record (ls :<: ts), Record (ls :>: ts))
```

Example

```
r' : (Record [("nombre", String)], Record [("edad", Int)])
r' = project ["nombre"] r
-- ({ "nombre" = "Juan" }, { "edad" = 20 })
```

Otras operaciones (3)

Left Union

Toma dos records y retorna la unión por izquierda de ambos.

Definition

```
(.||.) : DecEq lty => {ts, us : List (lty, Type)} ->
         Record ts -> Record us -> Record (ts :||: us)
```

Example

```
r' :: Record [("nombre", String), ("apellido", String)]
r' = ("nombre" .=. "Juan") .*. ("apellido" .=. "Torres") .*.
     emptyRec

r'' : Record [("nombre", String), ("edad", Int),
       ("apellido", String)]
r' = r .||. r''
-- { "nombre" = "Juan", "edad" = 20, "apellido" = "Torres" }
```

Otras operaciones (4)

Update

Modifica el valor de un campo en particular.

Definition

```
updateR : DecEq lty => {ts : List (lty, Type)} ->
          (l : lty) -> ty -> Record ts ->
          {auto hasF : HasField l ty ts} -> Record ts
```

Example

```
r' : Record [("nombre", String), ("edad", Int)]
r' = updateR "edad" 15 r
-- { "nombre" = "Juan", "edad" = 15 }
```

Caso de estudio (1)

Se realizó un caso de estudio más elaborado donde se utilizaron los records extensibles desarrollados en este trabajo. El caso consiste en modelar un pequeño lenguaje de expresiones aritméticas dado por:

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

Example

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

Caso de estudio (2)

Los records extensibles se utilizaron para modelar y mantener el ambiente con las variables y sus valores. Este ambiente es usado al momento de evaluar las expresiones.

Caso de estudio (2)

Los records extensibles se utilizaron para modelar y mantener el ambiente con las variables y sus valores. Este ambiente es usado al momento de evaluar las expresiones.

Definition (Definición del ambiente)

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

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

Definition (Definición del evaluador)

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

interp : Exp [] -> Nat
```

Caso de estudio (3)

Example (Ejemplos de construcción y evaluación de expresiones)

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

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

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

Conclusiones del trabajo (1)

Se cumplieron los objetivos propuestos para este trabajo:

- Solución expuesta como biblioteca de usuario.
- Typesafe.
- Records extensibles como first-class citizens.
- Cubre funcionalidades y operaciones de records extensibles.
- Práctica de usar.

Conclusiones del trabajo (2)

Otras conclusiones:

- Desarrollo en lenguajes con tipos dependientes es relativamente directo y sencillo.
- Las mayores dificultades se encuentran en probar teoremas/lemas, no en el desarrollo mismo.
- Se necesita tener un buen diseño de tipos y predicados.

Trabajo a futuro

En cuanto al trabajo, se identificaron varios aspectos para mejorar en el futuro:

- Implementar resto de operaciones sobre records extensibles.
- Parametrizar el record por un tipo abstracto, y no una lista ordenada.
- Estudiar replicar este trabajo en Agda.
- Agregar más tipos de expresiones al caso de estudio. Booleanos, if-then-else, etc.