Extensible records in Idris

Gonzalo Waszczuk, Alberto Pardo, Marcos Viera

Instituto de Computación, Universidad de la República
Montevideo, Uruguay

August 21th, 2017

Records

Data structure composed of a collection of fields, possibly with different types and with an associated label.

Records

Data structure composed of a collection of fields, possibly with different types and with an associated label.

Example (Record)

```
type Person = Record { surname : String, age : Int }
```

Extensible Record

Record that can be extended dynamically, adding new values with corresponding labels.

Extensible Record

Record that can be extended dynamically, adding new values with corresponding labels.

Example (Definition of record)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond", age = 30 }
```

Extensible Record

Record that can be extended dynamically, adding new values with corresponding labels.

Example (Definition of record)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond", age = 30 }
```

Example (Extension of record)

```
p' : Record { name : String, surname : String, age : Int }
p' = { name = "James" } .*. p
```

Extensible Record

Record that can be extended dynamically, adding new values with corresponding labels.

Example (Definition of record)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond", age = 30 }
```

Example (Extension of record)

```
p' : Record { name : String, surname : String, age : Int }
p' = { name = "James" } .*. p
```

Example (Alternative definition)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond" } .*.
    { age = 30 } .*.
    emptyRec
```

Other operations (1)

Example (Lookup)

```
n : String
n = p .!. surname
-- "Bond"
```

Other operations (1)

Example (Lookup)

```
n : String
n = p .!. surname
-- "Bond"
```

Example (Update)

```
p' : Record { surname : String, age : Int }
p' = updateR surname "Dean" p
-- { surname = "Dean", age = 30 }
```

Other operations (2)

Example (Delete)

```
p' : Record { age : Int }
p' = p .//. surname
-- { age = 30 }
```

Other operations (2)

Example (Delete)

```
p' : Record { age : Int }
p' = p .//. surname
-- { age = 30 }
```

Example (Union)

```
p' : Record { surname : String, name : String }
p' = { surname = "Bond", name = "James" }

p'' : Record { surname : String, age : Int, name : String}
p'' = p .||. p'
-- { surname = "Bond", age = 30, name = "James" }
```

Why use extensible records?

- Dynamic data structure. Can be extended, can be reduced.
- Fields can have arbitrary and variable types.
- Statically typed. The type of each individual field is known.
- Examples of uses: Language interpreter, database queries, configuration files, etc.

What are we looking for?

We are looking for a solution for extensible records that:

- Is typesafe
- Covers all operations and functionalities of extensible records.
- Is defined as a library.
- Has extensibles records as first-class citizens.
- Is practical to use.

What are we looking for?

We are looking for a solution for extensible records that:

- Is typesafe
- Covers all operations and functionalities of extensible records.
- Is defined as a library.
- Has extensibles records as first-class citizens.
- Is practical to use.

Haskell's HList library fulfills these functionalities. This work is inspired by HList, adapting it to dependent types.

Idris

- This solution is based on Idris.
- Idris is a functional, statically-typed, total, and dependently-typed language.
- In a dependently-typed language, types can depend on values and are first-class citizens.

Records as heterogeneous lists

Example (Original example)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond" } .*.
    { age = 30 } .*.
    emptyRec
```

Records as heterogeneous lists

Example (Original example)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond" } .*.
    { age = 30 } .*.
    emptyRec
```

An extensible record can be seen as a variable list of fields. These fields have different types, meaning the record can be represented with a *heterogeneous list*.

Records as heterogeneous lists

Example (Original example)

```
p : Record { surname : String, age : Int }
p = { surname = "Bond" } .*.
    { age = 30 } .*.
    emptyRec
```

An extensible record can be seen as a variable list of fields. These fields have different types, meaning the record can be represented with a *heterogeneous list*.

Example (Record as a heterogeneous list)

```
p = { surname = "Bond" } ::
    { age = 30 } ::
    Nil
```

Heterogeneous lists in Idris

Definition (HList in Idris)

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

Heterogeneous lists in Idris

Definition (HList in Idris)

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

Definition (Operation on HList)

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

Record defined with HList

Example

```
p : HList [String, Int]
p = "Bond" :> 30 :> HNil
```

Labelled HList

Where are the labels? This definition is incomplete.

Labelled HList

Where are the labels? This definition is incomplete.

Definition (Labelled HList)

```
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
```

Labelled HList

Where are the labels? This definition is incomplete.

Definition (Labelled HList)

```
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 defined with LHList)

```
p : LHList [("surname", String), ("age", Int)]
p = ("surname" .=. "Bond") :>
    ("age" .=. 30) :>
    HNil
```

Repeated labels

Example (Record with repeated labels)

```
p : LHList [("surname", String), ("surname", Int)]
p = ("surname" .=. "Bond") :>
    ("surname" .=. 30) :>
    HNil
```

Repeated labels

Example (Record with repeated labels)

```
p : LHList [("surname", String), ("surname", Int)]
p = ("surname" .=. "Bond") :>
    ("surname" .=. 30) :>
    HNil
```

This case compiles, but is incorrect. The definition of extensible records should not allow the construction of records with repeated labels.

Complete definition of extensible record

An extensible record is an heterogeneous list of fields with labels, such that no label is repeated.

Complete definition of extensible record

An extensible record is an heterogeneous list of fields with labels, such that no label is repeated.

Definition (Definition of extensible records)

```
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

Inductive predicate over lists that indicates that there are no repeated values.

IsSet

Inductive predicate over lists that indicates that there are no repeated values.

Definition (Definition of IsSet)

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

IsSet

Inductive predicate over lists that indicates that there are no repeated values.

Definition (Definition of IsSet)

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

`Elem` is an inductive predicate that indicates that a value belongs to a list.

Definition (Definition of Elem)

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

Final construction of a record (1)

With these definitions we can construct a record:

Example (Example of an extensible record)

```
r : Record [("surname", String), ("age", Int)]
r = MkRecord (IsSetCons not1 (IsSetCons not2 IsSetNil))
    (("surname" .=. "Bond") :>
    ("age" .=. 30) :>
    HNil)

not1 : Not (Elem "surname" ["age"])
not1 Here impossible

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

Final construction of a record (2)

Example (Empty record)

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

Extension of a record

The extension of a record simply verifies that the label is not repeated, and then adds the field to the list of fields of the record.

Extension of a record

The extension of a record simply verifies that the label is not repeated, and then adds the field to the list of fields of the 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)
```

Extension of a record

The extension of a record simply verifies that the label is not repeated, and then adds the field to the list of fields of the 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 (Example of an extensible record)

```
r : Record [("surname", String), ("age", Int)]
r = consR ("surname" .=. "Bond") not1 $
    consR ("age" .=. 30) not2 $
    emptyRec
```

Automatic generation of proofs (1)

The previous definition has a problem. It is always necessary to provide the proof of a label not being repeated.
We can improve it by automatically generating the proof at compilation time.

Automatic generation of proofs (2)

To be able to generate the proofs, we can use the fact that belonging to a list is a decidable predicate.
A predicate is decidable if you can always get a proof of it or its negation.

Automatic generation of proofs (2)

To be able to generate the proofs, we can use the fact that belonging to a list is a decidable predicate.
A predicate is decidable if you can always get a proof of it or its negation.

Definition (Definition of decidabilty in 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)
```

Automatic generation of proofs (2)

To be able to generate the proofs, we can use the fact that belonging to a list is a decidable predicate.
A predicate is decidable if you can always get a proof of it or its negation.

Definition (Definition of decidabilty in 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 (Decidability function of Elem)

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

Automatic generation of proofs (3)

We can use the decidability of a predicate to get a proof of it or its negation in compilation time. Depending on the result, we can unify it with our expected type or unify it with unit.

Automatic generation of proofs (3)

We can use the decidability of a predicate to get a proof of it or its negation in compilation time. Depending on the result, we can unify it with our expected type or unify it with 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
```

New definition of record extension (1)

With this new technique, we can redefine consR so that it automatically generates the proof of the label not belonging to the record.

New definition of record extension (1)

With this new technique, we can redefine consR so that it automatically generates the proof of the label not belonging to the record.

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))
```

New definition of record extension (2)

Example (Original example with new extension operator)

```
r : Record [("surname", String), ("age", Int)]
r = ("surname" .=. "Bond") .*.
    ("age" .=. 30) .*.
    emptyRec
```

Other operations

Other operations on records are also implemented:

- Lookup
- Update
- Append
- Project
- LeftUnion
- Delete
- DeleteLabels

Conclusions (1)

The principal objectives of this work were achieved:

- The solution is exposed as an user library.
- It's typesafe.
- Extensible records are first-class citizens.
- It covers the functionalities and operations of extensible records.
- It's relatively practical to use.

Conclusions (2)

Other conclusions:

- Development in dependently-typed languages is relatively direct and easy.
- Most dificulties are found in theorem proving, not in the development itself.
- A good type and predicate design is needed.

Future Work

We identified various aspects of this library to improve in the future:

- Implement the rest of the operations on extensible records.
- Parametrize the record by a sorted list or abstract type, not an ordered list.
- Replicate this library in Agda

Thanks!

GitHub: `https://github.com/gonzaw/extensible-records`

Gonzalo Waszczuk: gonzaw308@gmail.com
Alberto Pardo: pardo@fing.edu.uy
Marcos Viera: mviera@fing.edu.uy