Evolving datatypes

Monadic Warsaw

Andres Löh

10 January 2017

Well-Typed
The Haskell Consultants

Motivation

Datatypes evolve.

Motivation

Datatypes evolve.

Example:

```
data User = User
  { login     :: String
  , fullname  :: String
  , location  :: String
  }
```

Motivation

Datatypes evolve.

Example:

```
data User = User
  { login    :: String
  , fullname :: String
  }
```

Well-Typed

Motivation

Datatypes evolve.

Example:

```
data User = User
  { login     :: String
  , fullname  :: String
  , languages :: String
  }
```

Motivation

Datatypes evolve.

Example:

```
data User = User
  { login     :: String
  , fullname  :: String
  , languages :: [Language]
  }
```

Why is it a problem?

Within the program itself, it usually is not.

But programs communicate, and produce external representations of data:

- binary encodings,
- JSON,
- database entries,
- …

Different versions

External representations change ...

First version:

```
{ "login"    : "andres"
, "fullname" : "Andres Löh"
, "location" : "Regensburg"
}
```

Different versions

External representations change . . .

First version:

```
{ "login"    : "andres"
, "fullname" : "Andres Löh"
, "location" : "Regensburg"
}
```

"Current" version:

```
{ "login"     : "andres"
, "fullname"  : "Andres Löh"
, "languages" : ["Haskell", "Idris", "Agda"]
}
```

Well-Typed

Different versions

External representations change . . .

First version:

```
{ "login"    : "andres"
, "fullname" : "Andres Löh"
, "location" : "Regensburg"
}
```

"Current" version:

```
{ "login"     : "andres"
, "fullname"  : "Andres Löh"
, "languages" : ["Haskell", "Idris", "Agda"]
}
```

Program should be able to cope with both inputs.

[Some of the] Available Haskell options

safecopy

- Define all versions as separate Haskell datatypes.
- Define migration functions between the versions.
- Instantiate a class to get a versioned binary decoding.

[Some of the] Available Haskell options

safecopy

- Define all versions as separate Haskell datatypes.
- Define migration functions between the versions.
- Instantiate a class to get a versioned binary decoding.

api-tools

- Use a DSL to describe the changes between versions.
- Use Template Haskell to derive versioned decoders.

Use datatype-generic programming

Idea of datatype-generic programming

- Datatypes are given a uniform, structural, representation.
- We can convert between the original datatype and its representation.

Well-Typed

Idea of datatype-generic programming

- Datatypes are given a uniform, structural, representation.
- We can convert between the original datatype and its representation.
- We can define functions based on the representation that work on many different datatypes.

Idea of datatype-generic programming

- Datatypes are given a uniform, structural, representation.
- We can convert between the original datatype and its representation.
- We can define functions based on the representation that work on many different datatypes.
- Many different flavours of datatype-generic programming: GHC generics, Scrap your Boilerplate, uniplate, generics-sop, multirec, compdata, RepLib ...

Idea of datatype-generic programming

- Datatypes are given a uniform, structural, representation.
- We can convert between the original datatype and its representation.
- We can define functions based on the representation that work on many different datatypes.
- Many different flavours of datatype-generic programming: GHC generics, Scrap your Boilerplate, uniplate, generics-sop, multirec, compdata, RepLib ...

Representing types

```haskell
data User = User
  { login     :: Text
  , fullname  :: Text
  , languages :: [Language]
  }
```

Representing types

```
data User = User
  { login     :: Text
  , fullname  :: Text
  , languages :: [Language]
  }
```

```
type instance Code User = '['[Text, Text, [Language]]]
Rep (Code User) ~ SOP I (Code User)
```

Well-Typed

Representing types

```
data User = User
  { login     :: Text
  , fullname  :: Text
  , languages :: [Language]
  }
```

```
type instance Code User = '['[Text, Text, [Language]]]
Rep (Code User) ~ SOP I (Code User)
```

```
type family Code (a :: Type) :: [[Type]]
class Generic a where
  from :: a -> Rep (Code a)
  to   :: Rep (Code a) -> a
```

What is Rep?

```haskell
data User = User
  { login     :: Text
  , fullname  :: Text
  , languages :: [Language]
  }
```

Well-Typed

What is `Rep`?

```haskell
data User = User
  { login     :: Text
  , fullname  :: Text
  , languages :: [Language]
  }
```

```haskell
type instance Code User = '['[Text, Text, [Language]]]
```

Value of type `User`:

```haskell
User "andres" "Andres Löh" [Haskell, Idris, Agda]
```

Value of type `Rep (Code User)` (modulo syntactic clutter):

```haskell
C₀ ["andres", "Andres Löh", [Haskell, Idris, Agda]]
```

Well-Typed

Sums of products

```
SOP I xss ≈ NS (NP I) xss
```

```
data NS (f :: k -> Type) (xs :: [k]) where
  Z :: NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

```
data NP (f :: k -> Type) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)
```

Generic functions

```
class Encode a where
  encode  :: a -> [Bit]
  decoder :: Decoder a
```

Generic functions

```
class Encode a where
  encode  :: a -> [Bit]
  decoder :: Decoder a
```

Defined via induction on the representation:

```
gencode :: (Generic a, All2 Encode (Code a))
        => a -> [Bit]
gencode = ...
```

```
gdecoder :: (Generic a, All2 Encode (Code a))
         => Decoder a
gdecoder = ...
```

Yields defaults for the `Encode` class methods.

History of a datatype

```
User₁

User₂

User₃

User₄
```

Well-Typed

History of a datatype

```
User₁  Code User₁

User₂  Code User₂

User₃  Code User₃

User₄  Code User₄
```

Well-Typed

History of a datatype

```
User_1   Code User_1
    Migration (Code (User_1)) (Code (User_2))
User_2   Code User_2
    Migration (Code (User_2)) (Code (User_3))
User_3   Code User_3
    Migration (Code (User_3)) (Code (User_4))
User_4   Code User_4
```

```
data Migration :: [[Type]] -> [[Type]] -> Type where
  Migration :: (Rep a -> Rep b) -> Migration a b
```

History of a datatype

```
       Code User_1
    Migration (Code (User_1)) (Code (User_2))
       Code User_2
    Migration (Code (User_2)) (Code (User_3))
       Code User_3
    Migration (Code (User_3)) (Code (User_4))
User   Code User
```

```
data Migration :: [[Type]] -> [[Type]] -> Type where
  Migration :: (Rep a -> Rep b) -> Migration a b
```

History of a datatype

```
       Code User1
     Migration (Code (User1)) (Code (User2))
       Code User2
     Migration (Code (User2)) (Code (User3))
       Code User3
     Migration (Code (User3)) (Code (User4))
User   Code User
```

```
data Migration :: [[Type]] -> [[Type]] -> Type where
  Migration :: (Rep a -> Rep b) -> Migration a b

data History :: Version -> [[Type]] -> Type where
  Initial  :: History v c
  Revision :: (...)
           => Migration c' c
           -> History v' c'
           -> History v c
```

Simple migration

```
addConstructor :: Migration c ('[] ': c)
addConstructor = Migration shift
```

Simple migration

```
addConstructor :: Migration c ('[] ': c)
addConstructor = Migration shift
```

Good, but not quite satisfactory:

- By position rather than name.
- No way to actually give a name to a revision.

Include names in codes

```
data User = User {login :: String, fullname :: String}
```

Plain code:

```
type family Code (a :: Type) :: [[Type]]
type instance Code User =
  '['[String, String]]
```

Include names in codes

```
data User = User {login :: String, fullname :: String}
```

Plain code:

```
type family Code (a :: Type) :: [[Type]]
type instance Code User =
  '['[String, String]]
```

Code with metadata:

```
type MetaCode = [(Symbol, [(Symbol, Type)])]
type family Code' (a :: Type) :: MetaCode
type instance Code' User =
  '['("User", '['("login", String), '("fullname", String)])]
```

Stripping metadata:

```
type family Simplify (c :: MetaCode) :: [[Type]]
```

Well-Typed

Migrations based on codes with metadata

```
data Migration :: MetaCode
               -> MetaCode
               -> Type where
  Migration :: (Rep (Simplify a) -> Rep (Simplify b))
            -> Migration a b
```

Migrations based on codes with metadata

```
data Migration :: MetaCode
               -> MetaCode
               -> Type where
  Migration :: (Rep (Simplify a) -> Rep (Simplify b))
            -> Migration a b
```

```
addField :: (...)
         => Proxy (v :: Version)
         -> Proxy (d :: Symbol)   -- name of constructor
         -> Proxy (f :: Symbol)   -- name of field
         -> a                     -- default value
         -> History v' c
         -> History v (AddField d f c)
```

Example

```
personHistory :: History "4" (Code' User)
personHistory =
     changeType (Proxy @ "4")
       (Proxy @ "User") (Proxy @ "languages")
       parseLanguages
   $ addField (Proxy @ "3")
       (Proxy @ "User") (Proxy @ "languages")
       "Haskell"
   $ removeField (Proxy @ "2")
       (Proxy @ "User") (Proxy @ "location")
   $ initialRevision (Proxy @ "1")
       (Proxy@InitialCodeUser)
```

Well-Typed

Attaching histories to datatypes

```
class (Generic a, ...) => HasHistory a where
  type CurrentRevision a :: Symbol
  history :: Proxy a
          -> History (CurrentRevision a) (Code' a)
```

Encoding and decoding based on histories

```
hencode :: (HasHistory a, ...) => a -> [Bit]
```

- choose latest version from history
- encode version
- encode data generically

Encoding and decoding based on histories

```
hencode :: (HasHistory a, ...) => a -> [Bit]
```

- choose latest version from history
- encode version
- encode data generically

```
hdecode :: (HasHistory a, ...) => Decoder a
```

- decode version
- choose the corresponding version from history
- decode data generically for that version
- apply the remaining migration functions

Annoyances

For `hdecode`,

all types contained in all codes of all revisions

must be in the `Encode` class.

Annoyances

For `hdecode`,

all types contained in all codes of all revisions

must be in the `Encode` class.

This means:

- put class constraints in `History` type,
- index `History` over all intermediate versions,
- abstract `History` over class constraints.

Annoyances

For `hdecode`,

all types contained in all codes of all revisions

must be in the `Encode` class.

This means:

- put class constraints in `History` type,
- index `History` over all intermediate versions,
- abstract `History` over class constraints.

Also, versioning by datatype is actually not a good idea.

Conclusions

- Current code is proof of concept.
- New forms of migrations can be added.
- Not tied to a single encoding (i.e., different binary encodings, JSON, database, could all work with the same history).
- Comparatively much type safety.
- Also reverse migrations are possible.
- Efficiency?