Evolving datatypes
Monadic Warsaw

Andres Löh

10 January 2017
Datatypes evolve.
Datatypes evolve.

Example:

```haskell
data User = User
    { login :: String
    , fullname :: String
    , location :: String
    }
```
Datatypes evolve.

Example:

```haskell
data User = User
    { login :: String
  , fullname :: String
  }
```
Datatypes evolve.

Example:

```haskell
data User = User
  { login :: String
  , fullname :: String
  , languages :: String
  }
```
Datatypes evolve.

Example:

```haskell
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"}
```

Program should be able to cope with both inputs.

Well-Typed

Well-Typed
Different versions

External representations change …

First version:

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

“Current” version:

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

External representations change …

First version:

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

“Current” version:

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

Program should be able to cope with both inputs.
Available Haskell options

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

safecopy
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.
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

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

Well-Typed

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)
Representing types

```haskell
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
```

Well-Typed
What is Rep?

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

Value of type User: User "andres" "Andres Löh" [Haskell, Idris, Agda]
Value of type Rep (Code User) (modulo syntactic clutter):
C0["andres", "Andres Löh", [Haskell, Idris, Agda]].

Well-Typed

Well-Typed
What is **Rep**?

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

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

Value of type **User**:

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

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

\( C_0 ["andres", "Andres Löh", [Haskell, Idris, Agda]] \)
Sums of products

\[ SOP \ I \ xss \ \approx \ NS \ (NP \ I) \ xss \]

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

```haskell
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

Defined via induction on the representation:
gencode :: (Generic a, All2 Encode (Code a)) => a -> [Bit]
gdecoder :: (Generic a, All2 Encode (Code a)) => Decoder a

Yields defaults for the Encode class methods.
**Generic functions**

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

Defined via induction on the representation:

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

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

Yields defaults for the `Encode` class methods.
History of a datatype

User\textsubscript{1}

User\textsubscript{2}

User\textsubscript{3}

User\textsubscript{4}
## History of a datatype

<table>
<thead>
<tr>
<th>User</th>
<th>Code User</th>
</tr>
</thead>
<tbody>
<tr>
<td>User$_1$</td>
<td>Code User$_1$</td>
</tr>
<tr>
<td>User$_2$</td>
<td>Code User$_2$</td>
</tr>
<tr>
<td>User$_3$</td>
<td>Code User$_3$</td>
</tr>
<tr>
<td>User$_4$</td>
<td>Code User$_4$</td>
</tr>
</tbody>
</table>
data Migration :: [[Type]] -> [[Type]] -> Type where
  Migration :: (Rep a -> Rep b) -> Migration a b
History of a datatype

\[
\begin{align*}
\text{Code User}_1 & \\
\text{Migration} & \quad \text{(Code (User}_1\text{))} \quad \text{(Code (User}_2\text{))} \\
\text{Code User}_2 & \\
\text{Migration} & \quad \text{(Code (User}_2\text{))} \quad \text{(Code (User}_3\text{))} \\
\text{Code User}_3 & \\
\text{Migration} & \quad \text{(Code (User}_3\text{))} \quad \text{(Code (User}_4\text{))} \\
\text{User} & \quad \text{Code User} \\
\end{align*}
\]

\begin{verbatim}
data Migration :: [[Type]] -> [[Type]] -> Type where
  Migration :: (Rep a -> Rep b) -> Migration a b
\end{verbatim}
History of a datatype

Code User\textsubscript{1}
Migration (Code (User\textsubscript{1})) (Code (User\textsubscript{2}))

Code User\textsubscript{2}
Migration (Code (User\textsubscript{2})) (Code (User\textsubscript{3}))

Code User\textsubscript{3}
Migration (Code (User\textsubscript{3})) (Code (User\textsubscript{4}))

User \quad Code User

\underline{data} Migration :: [[Type]] \rightarrow [[Type]] \rightarrow Type \underline{where}
Migration :: (Rep a \rightarrow Rep b) \rightarrow Migration a b

\underline{data} History :: Version \rightarrow [[Type]] \rightarrow Type \underline{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

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

Plain code:

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

Code with metadata:

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

Stripping metadata:

```haskell
(type family) Simplify (c :: MetaCode) :: [[Type]]
```
Migrations based on codes with metadata

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

Well-Typed
Migrations based on codes with metadata

```haskell
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)
```

Well-Typed
Example

```haskell
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)
```

class (Generic a, ...) => HasHistory a where
  type CurrentRevision a :: Symbol
  history :: Proxy a
           -> History (CurrentRevision a) (Code‘ a)
hencode :: (HasHistory a, ...) => a -> [Bit]

- choose latest version from history
- encode version
- encode data generically
Encoding and decoding based on histories

\[
\text{hencode :: (HasHistory } a, \ldots) \Rightarrow a \rightarrow [\text{Bit}]
\]

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

\[
\text{hdecode :: (HasHistory } a, \ldots) \Rightarrow \text{Decoder } a
\]

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