Datatypes evolve.
Datatypes evolve.

Example:

```haskell
data Person = Person
    { name :: String
    , address :: String
    }
```
Motivation

Datatypes evolve.

Example:

```haskell
data Person = Person
  { name :: String }
```
Motivation

Datatypes evolve.

Example:

```haskell
data Person = Person
  {lastName :: String
   , firstName :: String
  }
```
Datatypes evolve.

Example:

```haskell
data Person = Person
  { lastName :: String,
    firstName :: String,
    years :: Int
  }
```
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:

```
{ "name" : "Aura Löh"
, "address" : "Regensburg"
}
```
External representations change …

First version:

```javascript
{"name" : "Aura Löh"
 , "address" : "Regensburg"
}
```

“Current” version:

```javascript
{"lastName" : "Löh"
 , "firstName" : "Aura"
 , "years" : 2
}
```
Different versions

External representations change . . .

First version:

```json
{
  "name" : "Aura Löh",
  "address" : "Regensburg"
}
```

“Current” version:

```json
{
  "lastName" : "Löh",
  "firstName" : "Aura",
  "years" : 2
}
```

Program should be able to cope with both inputs.
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.
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.
changes

version "0.4"
  changed record Person
  field added years :: Int

version "0.3"
  migration record Person SplitName

version "0.2"
  changed record Person
  field removed address

// initial version
version "0.1"
Use datatype-genericity
Migrations apply to different datatypes.

Serialization and deserialization to various formats are classic examples of datatype-generic programming.

Different versions of a datatype are usually closely related.
Representing types

data Person = Person
    { name :: String
    , address :: String
    }
Representing types

```haskell
data Person = Person
  { name :: String
  , address :: String
  }
```

```haskell
**type instance** Code Person = ' '[String, String]
Rep (Code Person) = SOP I (Code Person) ≈ Person
```
Representing types

data Person = Person
  { name :: String
  , address :: String
  }

(type instance) Code Person = '[]'[String, String]]
Rep (Code Person) = SOP I (Code Person) \sim Person

type family Code (a :: \ast) :: [[\ast]]
class Generic a where
  from :: a -> Rep (Code a)
  to :: Rep (Code a) -> a
What is Rep?

```haskell
data Person = Person
  { name :: String
  , address :: String
  }

type instance Code Person = '['[String, String]]
```

Value of type **Person**:

```
Person "Aura Löh" "Regensburg"
```

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

```
C₀ ['"Aura Löh", "Regensburg"]
```
Sums of products

SOP I xss ≈ NS (NP I) xss

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

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

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

Defined via induction on the representation:
```haskell
\text{gencode} ZZ \text{Hgeneric} a \text{L allR encode Hcode aII} > a \text{M} > \{\text{bit}\}
\text{gdecoder} ZZ Hgeneric a L allR encode Hcode a II decoder a
```

Yields defaults for the `encode` class methods.
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

Person$_1$

Person$_2$

Person$_3$

Person$_4$
History of a datatype

<table>
<thead>
<tr>
<th>Person</th>
<th>Code Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person$_1$</td>
<td>Code Person$_1$</td>
</tr>
<tr>
<td>Person$_2$</td>
<td>Code Person$_2$</td>
</tr>
<tr>
<td>Person$_3$</td>
<td>Code Person$_3$</td>
</tr>
<tr>
<td>Person$_4$</td>
<td>Code Person$_4$</td>
</tr>
</tbody>
</table>
History of a datatype

<table>
<thead>
<tr>
<th>Person _1</th>
<th>Code Person _1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration (Code (Person _1)) (Code (Person _2))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person _2</th>
<th>Code Person _2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration (Code (Person _2)) (Code (Person _3))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person _3</th>
<th>Code Person _3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration (Code (Person _3)) (Code (Person _4))</td>
<td></td>
</tr>
</tbody>
</table>

| Person \_4 | Code Person \_4 |

**data** Migration :: [[*]] -> [[*]] -> * **where**

Migration :: (Rep a -> Rep b) -> Migration a b
<table>
<thead>
<tr>
<th>Code Person</th>
<th>Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Person}_1$</td>
<td>$(\text{Code} (\text{Person}_1)) (\text{Code} (\text{Person}_2))$</td>
</tr>
<tr>
<td>$\text{Code Person}_2$</td>
<td>$(\text{Code} (\text{Person}_2)) (\text{Code} (\text{Person}_3))$</td>
</tr>
<tr>
<td>$\text{Code Person}_3$</td>
<td>$(\text{Code} (\text{Person}_3)) (\text{Code} (\text{Person}_4))$</td>
</tr>
</tbody>
</table>

**data Migration :: [[*]] -> [[*]] -> * where**

Migration :: (Rep a -> Rep b) -> Migration a b
History of a datatype

Code Person₁

Migration (Code (Person₁)) (Code (Person₂))

Code Person₂

Migration (Code (Person₂)) (Code (Person₃))

Code Person₃

Migration (Code (Person₃)) (Code (Person₄))

Person Code Person

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

**data** History :: Version -> [[*]] -> * **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

```haskell
data Person = Person {name :: String, address :: String}

Plain code:

`type family Code (a :: *) :: [[*]]`
`type instance Code Person = '['[String, String]]`
Include names in codes

```haskell
data Person = Person {name :: String, address :: String}

Plain code:

type family Code (a :: *) :: [[*]]
type instance Code Person = '[:['[String, String]]

Code with metadata:

type family Code’ (a :: *) :: [(Symbol, [(Symbol, *)])]
type instance Code’ Person = '[:('"Person", '[:("name", String), ('"address", String))]]

Stripping metadata:

type family Simplify (c :: [(Symbol, [(Symbol, *)])]) :: [[*]]
Migrations based on codes with metadata

```haskell
data Migration :: [(Symbol, [(Symbol, *)])]
  -> [(Symbol, [(Symbol, *)])]
  -> * where
Migration :: (Rep (Simplify a) -> Rep (Simplify b))
  -> Migration a b
```

Well-Typed
Migrations based on codes with metadata

```haskell
data Migration :: ([Symbol, [(Symbol, *)]]]
  -> [[Symbol, [(Symbol, *)]]]
  -> * where
Migration :: (Rep (Simplify a) -> Rep (Simplify b))
  -> Migration a b
```

```haskell
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)
```
personHistory :: History "0.4" (Code’ Person)
personHistory =
  addField [pr|"0.4"|]
  [pr|"Person"|] [pr|"years"|]
  (2 :: Int)
$ replaceField [pr|"0.3"|]
  [pr|"Person"|] [pr|"name"|]
  [pr|'["lastName", "firstName"]]
  splitName
$ removeField [pr|"0.2"|]
  [pr|"Person"|] [pr|"address"|]
$ initialRevision [pr|"0.1"|]
class (Generic a, ...) => HasHistory a where
  type CurrentRevision a :: Symbol
  history :: Proxy a
    -> History (CurrentRevision a) (Code' a)
Encoding and decoding based on histories

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

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

Well-Typed
Encoding and decoding based on histories

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

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

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

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

For `hdecode`, all types contained in all codes of all revisions must be in the `Encode` class.
An annoying detail

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.
Conclusions

- Current code is proof of concept.
- Implementing the migration steps (e.g. `addField`) is really ugly and a lot of work.
- But it works and is more safe than other approaches.
- Extends to nested versioning.
- Not tied to a single encoding.
- Efficiency?
- Future: writing older versions.