Write one program, get two (or three, or many)

BOB 2017, Berlin

Andres Löh

24 February 2017
Motivation
class ToJSON a where
  toJSON :: a -> Value
  toEncoding :: a -> Encoding

class FromJSON a where
  parseJSON :: Value -> Parser a
data Talk = MkTalk
  { talkNr    :: Int
   , talkAuthor :: Text
   , talkTitle :: Text
   , talkTrack :: Track
  }

data Track = Regular | Workshop
data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }

data Track = Regular | Workshop

thisTalk = MkTalk
    12
    "Andres Löh"
    "Write one program, get two (or three, or many)"
    Regular
Example instances

JSON representation

```haskell
data Track = Regular | Workshop

instance ToJSON Track where
  toJSON Regular = "Regular"
  toJSON Workshop = "Workshop"

instance FromJSON Track where
  parseJSON =
    withText "category" $ \txt ->
      if       txt == "Regular"  then pure Regular
      else if  txt == "Workshop" then pure Workshop
      else    fail "unknown category"
```

Well-Typed
Example instances

JSON representation

```haskell
data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }
```
Example instances

JSON representation

```haskell
data Talk = MkTalk
    { talkNr :: Int,
      , talkAuthor :: Text
      , talkTitle :: Text
      , talkTrack :: Track
    }

instance ToJSON Talk where
    toJSON (MkTalk nr author title cat) =
      object
        [ "nr" .= nr
        , "author" .= author
        , "title" .= title
        , "category" .= cat
        ]
```
Example instances

JSON representation

```haskell
data Talk = MkTalk
  { talkNr :: Int,
    talkAuthor :: Text,
    talkTitle :: Text,
    talkTrack :: Track
  }

instance FromJSON Talk where
  parseJSON =
    withObject "talk" $ \obj ->
      MkTalk
      <$> obj .: "nr"
      <*> obj .: "author"
      <*> obj .: "title"
      <*> obj .: "category"
```
Desired round-trip property

JSON representation

parseMaybe parseJSON (toJSON x) = Just x

Or:

decode (encode x) = Just x
Desired round-trip property

JSON representation

\[
\text{parseMaybe parseJSON (toJSON x)} = \text{Just } x
\]

Or:

\[
\text{decode (encode x)} = \text{Just } x
\]

Example:

\[
\text{GHCi}\text{> decode (encode thisTalk)} == \text{Just thisTalk}
\]

True
Not just JSON
class Binary t where
    put :: t -> Put
    get :: Get t
Example instances

Binary serialization

```haskell
data Track = Regular | Workshop
```

Well-Typed
Example instances

Binary serialization

```
data Track = Regular | Workshop

instance Binary Track where
    put Regular  = putWord8 0
    put Workshop = putWord8 1
```
Example instances
Binary serialization

```
data Track = Regular | Workshop

instance Binary Track where
  put Regular  = putWord8 0
  put Workshop = putWord8 1

get = do
  i <- getWord8
  case i of
    0  -> return Regular
    1  -> return Workshop
    _  -> fail "out of range"
```
Example instances

Binary serialization

data Talk = MkTalk
{ talkNr :: Int
, talkAuthor :: Text
, talkTitle :: Text
, talkTrack :: Track
}
Example instances

Binary serialization

data Talk = MkTalk
    { talkNr :: Int,
    , talkAuthor :: Text,
    , talkTitle :: Text,
    , talkTrack :: Track
    }

instance Binary Talk where
    put (MkTalk nr author title cat) =
        put nr >> put author >> put title >> put cat
    get =
        MkTalk <$> get <*> get <*> get <*> get
Desired round-trip property
Binary serialization

\[
\text{runGet get (runPut (put x)) = x}
\]

Or:

\[
\text{decode (encode x) = x}
\]
Desired round-trip property

Binary serialization

\[
\text{runGet } \text{get} \ (\text{runPut } \ (\text{put } x)) = x
\]

Or:

\[
\text{decode } \ (\text{encode } x) = x
\]

Example:

\[
\text{GHCi}> \ \text{decode } \ (\text{encode thisTalk}) == \text{thisTalk}\\
\text{True}
\]
Other similar examples

SQL database table rows:

```haskell
class ToRow a where
  toRow :: a -> [Action]

class FromRow a where
  fromRow :: RowParser a
```

Well-Typed
Other similar examples

SQL database table rows:

```haskell
class ToRow a where
toRow :: a -> [Action]
```

```haskell
class FromRow a where
fromRow :: RowParser a
```

Textual representation:

```haskell
class Show a where
showsPrec :: Int -> a -> ShowS
```

```haskell
class Read a where
readsPrec :: Int -> ReadS a
```
We write (at least) two programs.
The programs contain the same (very similar) information.
There are desired properties that are easily violated.
(Datatype-)Generic Programming
Derive everything automatically

- deriving instance Generic Talk
- deriving instance Generic Track
Derive everything automatically

**deriving instance** Generic Talk
**deriving instance** Generic Track

**instance** ToJSON Talk
**instance** ToJSON Track
**instance** FromJSON Talk
**instance** FromJSON Track
**instance** Binary Talk
**instance** Binary Track
Write no program, get many?

The datatype is a program!

Programs follow the structure of the datatypes precisely. This is not always good.

Well-Typed
Write no program, get many?

- The datatype is a program!
Write no program, get many?

- The datatype is a program!
- Programs follow the structure of the datatypes precisely.
- This is not always good.
Disadvantages of generic programming

- External representations are implicit.
- And under the control of (third-party) library authors.
- Limited flexibility.
All or nothing?

Either:

▶ Use the derived instances.
▶ Enjoy the lack of boilerplate.
▶ Possibly live with a suboptimal external (or internal) representation.

Or:

▶ Write instances yourself.
▶ Stay in control.
▶ Lots of hand-written, error-prone code with subtle proof obligations.
Is there another option?
What if there are different requirements?

```
{ "nr": 12,
  "author": "Andres Löh",
  "title": "Write one program, get two (or three, or many)",
  "category": "Regular"
}

vs.

{ "nr": 12,
  "author": "Andres Löh",
  "title": "Write one program, get two (or three, or many)",
  "is-workshop": false
}
```
A single description for both (all) desired functions:

```haskell
instance Json Talk where
  grammar =
    fromPrism _Talk
    . object
      ( prop "nr"
        . prop "name"
        . prop "title"
        . prop "category"
      )

instance Json Track where
  grammar =
    fromPrism _Regular . "Regular"
 <> fromPrism _Workshop . "Workshop"
```
A single description

- Explicit. Can be different from datatype.
- Still strongly typed.
- Easy to adapt.
Switching representations

```haskell
instance Json Talk where
  grammar =
    fromPrism _Talk
      . object
       ( prop "nr"
          . prop "name"
          . prop "title"
          . prop "category"
       )

instance Json Track where
  grammar =
    fromPrism _Regular . "Regular"
    <> fromPrism _Workshop . "Workshop"
```

Well-Typed
Switching representations

```haskell
instance Json Talk where
  grammar =
    fromPrism _Talk
    . object
      ( prop "nr"
        . prop "name"
        . prop "title"
        . property "is-workshop" boolTrack
      )

boolTrack =
  fromPrism _Regular . false
<> fromPrism _Workshop . true
```

Well-Typed
Switching representations

```
instance Json Talk where
  grammar =
    fromPrism _Talk
  . object
    ( prop "nr"
    . prop "name"
    . prop "title"
    . ( property "is-workshop" boolTrack
      <> defaultValue Regular
    )
    )

boolTrack =
  fromPrism _Regular . false
<> fromPrism _Workshop . true
```
A closer look
A prism generalizes a Haskell constructor.
Combines a constructor function with a compatible matcher.
Prisms

- A prism generalizes a Haskell constructor.
- Combines a constructor function with a compatible matcher.

\[
\text{stackPrism} :: (a \to b) \to (b \to \text{Maybe } a) \\
\quad \to \text{StackPrism } a \ b
\]

\[
\text{forward} :: \text{StackPrism } a \ b \to (a \to b)
\]

\[
\text{backward} :: \text{StackPrism } a \ b \to (b \to \text{Maybe } a)
\]
Prisms

- A prism generalizes a Haskell constructor.
- Combines a constructor function with a compatible matcher.

```
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b
forward    :: StackPrism a b -> (a -> b)
backward   :: StackPrism a b -> (b -> Maybe a)
```

Laws:

- `backward p (forward p a) = Just a`
- `backward p b = Just a ⇒ forward p a = b`
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b

data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b

data Talk = MkTalk
    {talkNr :: Int
    ,talkAuthor :: Text
    ,talkTitle :: Text
    ,talkTrack :: Track
    }

MkTalk :: Int -> Text -> Text -> Track -> Talk
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b

data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }

MkTalk :: Int -> Text -> Text -> Track -> Talk
    (Int, Text, Text, Track) -> Talk
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b

data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }

MkTalk :: Int -> Text -> Text -> Track -> Talk
    (Int, Text, Text, Track) -> Talk
    (Int, (Text, (Text, (Track, ())))) -> Talk
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b

data Talk = MkTalk
    { talkNr :: Int
    , talkAuthor :: Text
    , talkTitle :: Text
    , talkTrack :: Track
    }

MkTalk :: Int -> Text -> Text -> Track -> Talk
        (Int, Text, Text, Track) -> Talk
        (Int, (Text, (Text, (Track, s)))) -> Talk
**Stacks**

\[ \text{stackPrism} : (a \to b) \to (b \to \text{Maybe } a) \to \text{StackPrism } a \; b \]

```haskell
\textbf{data} \text{Talk} = \text{MkTalk}
  \{ \text{t}alk\text{Nr} :: \text{Int} \\
    , \text{t}alk\text{Author} :: \text{Text} \\
    , \text{t}alk\text{Title} :: \text{Text} \\
    , \text{t}alk\text{Track} :: \text{Track} \\
  \}
```

\[ \text{MkTalk} :: \text{Int} \to \text{Text} \to \text{Text} \to \text{Track} \to \text{Talk} \\
    (\text{Int}, \text{Text}, \text{Text}, \text{Track}) \to \text{Talk} \\
    (\text{Int}, (\text{Text}, (\text{Text}, (\text{Track}, ())))) \to \text{Talk} \\
    (\text{Int}, (\text{Text}, (\text{Text}, (\text{Track}, s)))) \to (\text{Talk}, s) \\
    (\text{Int} \text{ -- Text} \text{ -- Text} \text{ -- Track} \text{ -- s}) \to (\text{Talk} \text{ -- s}) \]
Example stack prisms

_Talk ::
  StackPrism
  (Int :- Text :- Text :- Track :- s) (Track :- s)

_True :: StackPrism s (Bool :- s)

_False :: StackPrism s (Bool :- s)

_Just :: StackPrism (a :- s) (Maybe a :- s)

_Pair :: StackPrism (a :- b :- s) ((a, b) :- s)

_Nil :: StackPrism s ([a] :- s)

_Cons :: StackPrism (a :- [a] :- s) ([a] :- s)
Example stack prisms

_Talk ::
   StackPrism
   (Int :- Text :- Text :- Track :- s) (Track :- s)

_False :: StackPrism s (Bool :- s)
_True :: StackPrism s (Bool :- s)
Example stack prisms

_Talk ::
    StackPrism
    (Int :- Text :- Text :- Track :- s) (Track :- s)

_False :: StackPrism s (Bool :- s)
_True :: StackPrism s (Bool :- s)

_Nothing :: StackPrism s (Maybe a :- s)
_Just :: StackPrism (a :- s) (Maybe a :- s)
Example stack prisms

_Talk ::
    StackPrism
    (Int :- Text :- Text :- Track :- s) (Track :- s)

_False :: StackPrism s (Bool :- s)
_True :: StackPrism s (Bool :- s)

_Nothing :: StackPrism s (Maybe a :- s)
_Just :: StackPrism (a :- s) (Maybe a :- s)

_Pair :: StackPrism (a :- b :- s) ((a, b) :- s)
Example stack prisms

_Talk ::
  StackPrism
  (Int :- Text :- Text :- Track :- s) (Track :- s)

_False :: StackPrism s (Bool :- s)
_True :: StackPrism s (Bool :- s)

_Nothing :: StackPrism s (Maybe a :- s)
_Just :: StackPrism (a :- s) (Maybe a :- s)

_Pair :: StackPrism (a :- b :- s) ((a, b) :- s)

_Nil :: StackPrism s ([a] :- s)
_Cons :: StackPrism (a :- [a] :- s) ([a] :- s)
Obtaining stack prisms

These can be derived mechanically:

- `PrismList (P _Talk) = mkPrismList :: StackPrisms Talk`
- `PrismList (P _Regular :& P _Workshop) = mkPrismList :: StackPrisms Track`

Works via datatype-generic programming:

- `mkPrismList :: (MkPrismList (Rep a), Generic a) => StackPrisms a`
Another look at the descriptions

```haskell
instance Json Talk where
    grammar =
        fromPrism _Talk
        . object
        ( prop "nr"
        . prop "name"
        . prop "title"
        . ( property "is-workshop" boolTrack
                <> defaultValue Regular
            )
        )

boolTrack =
    fromPrism _Regular  . false
<> fromPrism _Workshop  . true
```
Grammars

Also parameterized by stacks:

Grammar \( n \ a \ b \)

Here:

- \( n \) is the syntactic category,
- \( a \) is the “source” stack,
- \( b \) is the “target” stack.
Examples
Grammars

```
GHCi> :type fromPrism _Regular
fromPrism _Regular :: Grammar n a (Track :- a)
```
Examples
Grammars

GHCi> :type fromPrism _Regular
fromPrism _Regular :: Grammar n a (Track :- a)

GHCi> :type false
false :: Grammar Val (Value :- a) a
Examples

Grammars

GHCi> :type fromPrism _Regular
fromPrism _Regular :: Grammar n a (Track :- a)

GHCi> :type false
false :: Grammar Val (Value :- a) a

GHCi> :type fromPrism _Regular . false
... :: Grammar Val (Value :- b) (Track :- b)
Examples
Grammars

```
GHCi> :type fromPrism _Regular
fromPrism _Regular :: Grammar n a (Track :- a)

GHCi> :type false
false :: Grammar Val (Value :- a) a

GHCi> :type fromPrism _Regular . false
... :: Grammar Val (Value :- b) (Track :- b)

GHCi> gdecode (fromPrism _Regular . false) "false"
Just Regular
```
Examples

Grammars

GHCi> :type fromPrism _Regular
fromPrism _Regular :: Grammar n a (Track :- a)

GHCi> :type false
false :: Grammar Val (Value :- a) a

GHCi> :type fromPrism _Regular . false
... :: Grammar Val (Value :- b) (Track :- b)

GHCi> gdecode (fromPrism _Regular . false) "false"
  Just Regular

GHCi> gencode (fromPrism _Regular . false) Regular
  Just "false"
Composition:

\[
( . ) :: \text{Grammar } n \; b \; c \; \to \; \text{Grammar } n \; a \; b \; \to \; \text{Grammar } n \; a \; c
\]
Combinators

Grammars

Composition:

\((\cdot)\) :: Grammar n b c \rightarrow Grammar n a b \rightarrow Grammar n a c

Choice:

\((<>\cdot)\) :: Grammar n a b \rightarrow Grammar n a b \rightarrow Grammar n a b
class Json a where
  grammar :: Grammar Val (Value :- b) (a :- b)

genode ::
  Grammar Val (Value :- ()) (a :- ())
  -> a -> Maybe ByteString

gdecode ::
  Grammar Val (Value :- ()) (a :- ())
  -> ByteString -> Maybe a
The expectation is that:

\[
\text{gen} \ g \ a = \text{Just} \ b \Rightarrow \\
\text{gdecode} \ g \ b = \text{Just} \ a
\]
A final look at the descriptions

```haskell
instance Json Talk where
  grammar =
    fromPrism _Talk
      . object
        ( prop "nr"
          . prop "name"
          . prop "title"
            ( property "is-workshop" boolTrack
                <> defaultValue Regular
              )
        )
  
boolTrack =
  fromPrism _Regular . false
<> fromPrism _Workshop . true
```
Stepping back
What have we achieved?

- A better representation.
- Sufficient to compute multiple interpretations.
- Works for interpretations having different directions.
- Widely applicable?
This and other solutions

The code shown for JSON is based on:

JsonGrammar

by Martijn van Steenbergen
This and other solutions

The code shown for JSON is based on:

JsonGrammar
by Martijn van Steenbergen

The same idea (stack prisms, composition, DSL, interpretations) can be applied to other scenarios:

- binary serialization,
- SQL database table rows,
- human-readable textual representations,
- ...
Some other notable libraries

invertible-syntax
by Tillmann Rendel (also Haskell Symposium 2010 paper)

roundtrip, roundtrip-string, roundtrip-xml, roundtrip-aeson
by Stefan Wehr and David Leuschner
(roundtrip-aeson by Thomas Sutton and Christian Marie)

boomerang, web-routes-boomerang
by Jeremy Shaw
(where web-routes-boomerang is based on Zwaluw, by Sjoerd Visscher and (again) Martijn van Steenbergen)
servant
by Alp Mestanogullari, Sönke Hahn, Julian Arni and others
The more general message

- Choose suitable representations for your programs.
- If you write several programs that are interrelated in complicated ways, you are doing it wrong.
- Some scenarios in specific applications may be much easier (additional conventions and constraints).
Questions?