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

BOB 2017, Berlin

Andres Löh

24 February 2017

Well-Typed
The Haskell Consultants

Motivation

JSON representation

```haskell
class ToJSON a where
  toJSON     :: a -> Value
  toEncoding :: a -> Encoding
```

```haskell
class FromJSON a where
  parseJSON :: Value -> Parser a
```

Example datatype

JSON representation

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

Example datatype

JSON representation

```
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

```
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"
```

Example instances

JSON representation

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

Well-Typed

Example instances

JSON representation

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

```haskell
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
  }
```

```haskell
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

```
parseMaybe parseJSON (toJSON x) = Just x
```

Or:

```
decode (encode x) = Just x
```

Example:

```
GHCi> decode (encode thisTalk) == Just thisTalk
 True
```

Not just JSON

Binary serialization

```
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

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

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

Well-Typed

Example instances

Binary serialization

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

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

```haskell
  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

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

```haskell
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

```
runGet get (runPut (put x)) = x
```

Or:

```
decode (encode x) = x
```

Desired round-trip property

Binary serialization

```
runGet get (runPut (put x)) = x
```

Or:

```
decode (encode x) = x
```

Example:

```
GHCi> decode (encode thisTalk) == thisTalk
 True
```

Other similar examples

SQL database table rows:

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

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

Other similar examples

SQL database table rows:

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

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

Textual representation:

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

```
class Read a where
  readsPrec :: Int -> ReadS a
```

Common theme

- 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?

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
}
```

Well-Typed

The solution

A single description for both (all) desired functions:

```
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

```
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"
```

Switching representations

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

```haskell
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

Prisms

- 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.

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

Stacks

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

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

Stacks

```haskell
stackPrism :: (a -> b) -> (b -> Maybe a) -> StackPrism a b
```

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

```haskell
MkTalk :: Int -> Text -> Text -> Track -> Talk
```

Stacks

```
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
```

Well-Typed

Stacks

```
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
```

Stacks

```
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

```
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
          (Int, (Text, (Text, (Track, s)))) -> (Talk, s)
          (Int :- Text :- Text :- Track :- s) -> (Talk :- s)
```

Example stack prisms

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

```
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
```

Well-Typed

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)
```

Well-Typed

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"
```

Well-Typed

Composition:

```
(.) :: Grammar n b c -> Grammar n a b -> Grammar n a c
```

Composition:

```
(.) :: Grammar n b c -> Grammar n a b -> Grammar n a c
```

Choice:

```
(<>) :: Grammar n a b -> Grammar n a b -> Grammar n a b
```

```haskell
class Json a where
  grammar :: Grammar Val (Value :- b) (a :- b)
```

```haskell
gencode ::
      Grammar Val (Value :- ()) (a :- ())
   -> a -> Maybe ByteString

gdecode ::
      Grammar Val (Value :- ()) (a :- ())
   -> ByteString -> Maybe a
```

The expectation is that:

```
gencode g a = Just b ⇒
  gdecode g b = Just a
```

A final look at the descriptions

```
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
```

Well-Typed

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)

Type level

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?