Round tripping balls

(with partial isomorphisms & Haskell)

What do I want to show you?

How to write better printer/parsers such that we type less, think less and make fewer mistakes.

Outline

Outline

1. Define problem

Outline

1. Define problem
2. Summarise paper

Outline

1. Define problem
2. Summarise paper
3. Build your own

What are we fixing?

The problem

Writing isomorphic round-trip printer/parsers with the get/put idiom is redundant and error prone.

What are we fixing?

We can have either bouncy or lumpy balls

```
data Ball
    = Lumpy  { _colour      :: Text
             , _lumps       :: [[Bool]]
             }
    | Bouncy { _bouncyness :: Double }
  deriving (Eq, Show)
```

What are we fixing?

Bouncy balls are happy, lumpy ones are not.

```
[ Lumpy  {_colour = "Rainbow"
         , _lumps = [[True,False],[False,False]]}
, Bouncy {_bouncyness = 3.14159265358979}]
```

⬇

```
[
   {
      "colour" : "Rainbow",
      ":D" : false,
      "lumps" : [[true,false],[false,false]]
   },
   {
      "bouncyness" : 3.14159265358979,
      ":D" : true
   }
]
```

What are we fixing?

We want to parse (partial)

```
instance FromJSON Ball where
  parseJSON :: Value -> Parser Ball
  parseJSON (Object o) = do
    happy_ball <- o .: ":D"
    if happy_ball then parseBouncy else parseLumpy
   where
    parseBouncy =
        Bouncy <$> o .: "bouncyness"
    parseLumpy =
        Lumpy <$> o .: "colour" <*> o .: "lumps"
```

What are we fixing?

And we want to print.

```haskell
instance ToJSON Ball where
  toJSON :: Ball -> Value
  toJSON (Lumpy colour lump_map) =
      object [ ":D"      .= True
             , "colour"  .= colour
             , "lumps"   .= lump_map
             ]
  toJSON (Bouncy bouncyness) =
      object [ ":D"          .= True
             , "bouncyness" .= bouncyness
             ]
```

What are we fixing?

Duplicate information!

Potential errors and good programmers *HATE* typing

Why don't you. . .

- Just write some tests

Why don't you...

- ~~Just write some tests~~ Unnecessary boilerplate.
- Stop whining and trust the libraries

Why don't you...

- ~~Just write some tests~~ Unnecessary boilerplate.
- Stop whining and trust the libraries

Why don't you. . .

- ~~Just write some tests~~ Unnecessary boilerplate.
- ~~Stop whining and trust the libraries~~ Too flexible.

Why don't you...

- ~~Just write some tests~~ Unnecessary boilerplate.
- ~~Stop whining and trust the libraries~~ Too flexible.
- Use template haskell/generics

Why don't you. . .

- ~~Just write some tests~~ Unnecessary boilerplate.
- ~~Stop whining and trust the libraries~~ Too flexible.
- ~~Use template haskell/generics~~ Not flexible enough.

Introducing: Enterprise JSON

Introducing: Enterprise JSON

“datetime”

```
"27/6/2013 10:29 pm"
```

Introducing: Enterprise JSON

“datetime”

```
"27/6/2013 10:29 pm"
```

“date”

```
"12/12/2012"
```

Introducing: Enterprise JSON

“datetime”

```
"27/6/2013 10:29 pm"
```

“date”

```
"12/12/2012"
```

“mmdddate”

```
"12/12"
```

Introducing: Enterprise JSON

“datetime”

`"27/6/2013 10:29 pm"`

“date”

`"12/12/2012"`

“mmdddate”

`"12/12"`

“mmyydate”

`"12/2012"`

OR

`"12.2012"`

OR

`"122012"`

Introducing: Enterprise JSON

“checkbox”

```
"T"
```

Introducing: Enterprise JSON

“checkbox”

```
"T"
```

“currency”, “currency2” and “poscurrency”

```
"00.03"
```

Introducing: Enterprise JSON

“checkbox”

`"T"`

“currency”, “currency2” and “poscurrency”

`"00.03"`

“posfloat”, “nonnegfloat”

`"2.718281828459045"`

Introducing: Enterprise JSON

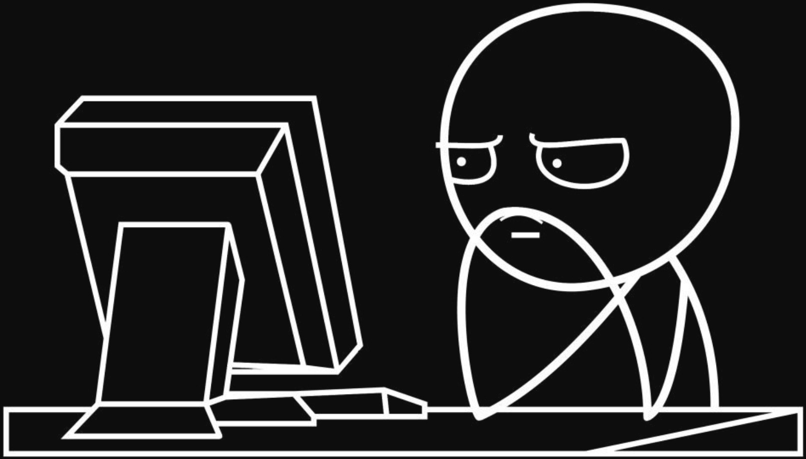
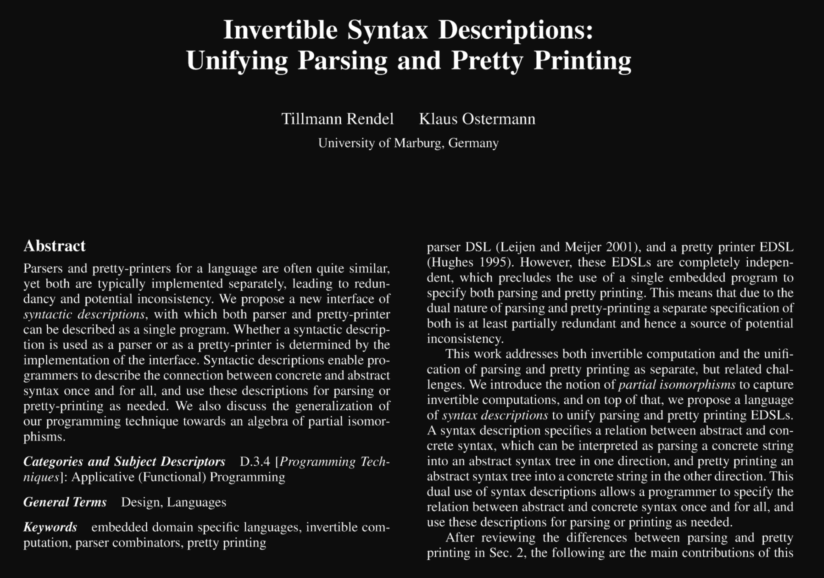


Figure 1: Concern for sanity

A wild paper appears!



Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing

Tillmann Rendel Klaus Ostermann

University of Marburg, Germany

Abstract

Parsers and pretty-printers for a language are often quite similar, yet both are typically implemented separately, leading to redundancy and potential inconsistency. We propose a new interface of *syntactic descriptions*, with which both parser and pretty-printer can be described as a single program. Whether a syntactic description is used as a parser or as a pretty-printer is determined by the implementation of the interface. Syntactic descriptions enable programmers to describe the connection between concrete and abstract syntax once and for all, and use these descriptions for parsing or pretty-printing as needed. We also discuss the generalization of our programming technique towards an algebra of partial isomorphisms.

Categories and Subject Descriptors D.3.4 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Design, Languages

Keywords embedded domain specific languages, invertible computation, parser combinators, pretty printing

parser DSL (Leijen and Meijer 2001), and a pretty printer EDSL (Hughes 1995). However, these EDSLs are completely independent, which precludes the use of a single embedded program to specify both parsing and pretty printing. This means that due to the dual nature of parsing and pretty-printing a separate specification of both is at least partially redundant and hence a source of potential inconsistency.

This work addresses both invertible computation and the unification of parsing and pretty printing as separate, but related challenges. We introduce the notion of *partial isomorphisms* to capture invertible computations, and on top of that, we propose a language of *syntax descriptions* to unify parsing and pretty printing EDSLs. A syntax description specifies a relation between abstract and concrete syntax, which can be interpreted as parsing a concrete string into an abstract syntax tree in one direction, and pretty printing an abstract syntax tree into a concrete string in the other direction. This dual use of syntax descriptions allows a programmer to specify the relation between abstract and concrete syntax once and for all, and use these descriptions for parsing or printing as needed.

After reviewing the differences between parsing and pretty printing in Sec. 2, the following are the main contributions of this

Invertible Syntax Descriptions: way of the get/put

Given a datatype:

```
data List a
  = Nil
  | Cons a (List a)
```

Invertible Syntax Descriptions: way of the get/put

Printing[1]

```
type Printer a = a -> Doc

printMany :: Printer a -> Printer (List a)
printMany p list
  = case list of
    Nil       -> text ""
    Cons x xs -> p x
              <> printMany p xs
```

Invertible Syntax Descriptions: way of the get/put

Printing[1]

```
type Printer a = a -> Doc

printMany :: Printer a -> Printer (List a)
printMany p list
  = case list of
    Nil       -> text ""
    Cons x xs -> p x
              <> printMany p xs
```

Parsing[2]

```
parseMany :: Parser a -> Parser (List a)
parseMany p
  =   const Nil <$> text ""
 <|> Cons       <$> p
                <*> parseMany p
```

Invertible Syntax Descriptions: way of the get/put

It would be nice if...

```
combined :: Unicorn x => x a -> x (List a)
combined p
  =  magic Nil <$> fairies ""
 <|> Cons      <$> p
               <*> parseMany p
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```haskell
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> Parser p = Parser $ (fmap . first) f . p
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> Parser p = Parser $ (fmap . first) f . p
```

Printer fmap

```
type Printer a = a -> Doc

(<$>) :: (a -> b) -> Printer a -> Printer b
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> Parser p = Parser $ (fmap . first) f . p
```

Printer fmap

```
type Printer a = a -> Doc

(<$>) :: (a -> b) -> Printer a -> Printer b
```

Can you implement this? ⇧

Invertible Syntax Descriptions: co/contravariance

Covariant ⇩

Parser fmap

```haskell
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
```

Printer fmap

```haskell
type Printer a = a -> Doc

(<$>) :: (b -> a) -> Printer a -> Printer b
```

Contravariant ⇧

Invertible Syntax Descriptions: co/contravariance

Partial Iso[3] (simplified)

```
data Iso a b = Iso
    { apply   :: a -> Maybe b
    , unapply :: b -> Maybe a
    }
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
```

Printer fmap

```
type Printer a = a -> Doc

(<$>) :: (b -> a) -> Printer a -> Printer b
```

Invertible Syntax Descriptions: co/contravariance

Parser fmap

```
newtype Parser a = Parser (String -> [(a, String)])

(<$>) :: (a -> b) -> Parser a -> Parser b
```

Printer fmap

```
type Printer a = a -> Doc

(<$>) :: (b -> a) -> Printer a -> Printer b
```

The solution: IsoFunctor[4]

```
class IsoFunctor f where
  (<$>) :: Iso a b -> f a -> f b
```

Invertible Syntax Descriptions: co/contravariance

The important things about partial isos and IsoFunctor:

Invertible Syntax Descriptions: co/contravariance

The important things about partial isos and IsoFunctor:

- Unifying a functor requires both $a \rightarrow b$ and $b \rightarrow a$

Invertible Syntax Descriptions: co/contravariance

The important things about partial isos and IsoFunctor:

- Unifying a functor requires both $a \to b$ and $b \to a$
- We unify both with a partial Iso, where these functions can fail

Invertible Syntax Descriptions: co/contravariance

The important things about partial isos and IsoFunctor:

- Unifying a functor requires both $a \rightarrow b$ and $b \rightarrow a$
- We unify both with a partial Iso, where these functions can fail
- We defined IsoFunctor (from partial isos to printer/parsers)

Invertible Syntax Descriptions: applicative

Normal applicative

```
(<*>) :: f (a -> b) -> f a -> f  b

instance Applicative Parser where
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

Invertible Syntax Descriptions: applicative

Normal applicative

```
(<*>) :: f (a -> b) -> f a -> f  b


instance Applicative Parser where
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

Adapting that directly

```
class UnhelpfulIsoApplicative where
  (<*>) :: f (Iso a b) -> f a -> f b
```

Invertible Syntax Descriptions: applicative

Normal applicative

```haskell
(<*>) :: f (a -> b) -> f a -> f  b

instance Applicative Parser where
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

Adapting that directly

```haskell
class UnhelpfulIsoApplicative where
  (<*>) :: f (Iso a b) -> f a -> f b
```

Falls apart on Printer (the contravariant one)

```haskell
type Printer a = a -> Doc

instance Applicative Printer where
  (<*>) :: (Iso a b -> Doc) -> (a -> Doc) -> b -> Doc
  (f <*> g) b = error "impossible!"
```

Invertible Syntax Descriptions: applicative

Normal applicative

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

Invertible Syntax Descriptions: applicative

Normal applicative

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

***#!@ it, associate right and tuple (ProductFunctor[5])**

```
class ProductFunctor f where
  infixr 6 <*>
  (<*>) :: f a -> f b -> f (a, b)
```

Invertible Syntax Descriptions: applicative

Normal (currying applicative, left associative)

```
f :: Applicative f
  => (a -> b -> c -> d)
  -> f a -> f b -> f c -> f d
f ctor fa fb fc =   ctor <$> fa  <*> fb  <*> fc
f ctor fa fb fc = ((ctor <$> fa) <*> fb) <*> fc
```

Invertible Syntax Descriptions: applicative

Normal (currying applicative, left associative)

```
f :: Applicative f
  => (a -> b -> c -> d)
  -> f a -> f b -> f c -> f d
f ctor fa fb fc =   ctor <$> fa  <*> fb  <*> fc
f ctor fa fb fc = ((ctor <$> fa) <*> fb) <*> fc
```

Our new, alternate universe

```
f :: (ProductFunctor f, IsoFunctor f)
  => Iso (a, (b, c)) d
  -> f a -> f b -> f c -> f d
f ctor fa fb fc = ctor <$>  fa <*> fb  <*> fc
f ctor fa fb fc = ctor <$> (fa <*> (fb <*> fc))
```

Invertible Syntax Descriptions: applicative

We want these tuple tree isos for our data types

```
nil  :: Iso ()           (List a)
cons :: Iso (a, List a) (List a)
```

Invertible Syntax Descriptions: applicative

We want these tuple tree isos for our data types

```
nil  :: Iso ()           (List a)
cons :: Iso (a, List a) (List a)
```

So we magic them up from the data type:

```
data List a
  = Nil
  | Cons a (List a)

defineIsomorphisms ''List
```

Invertible Syntax Descriptions: applicative

The important things about ProductFunctor:

Invertible Syntax Descriptions: applicative

The important things about ProductFunctor:

- Naively adapting Applicative leaves us with an uninhabitable type.

Invertible Syntax Descriptions: applicative

The important things about ProductFunctor:

- Naively adapting Applicative leaves us with an uninhabitable type.
- We use ProductFunctor, it has tuples instead of currying and associates right

Invertible Syntax Descriptions: applicative

The important things about ProductFunctor:

- Naively adapting Applicative leaves us with an uninhabitable type.
- We use ProductFunctor, it has tuples instead of currying and associates right
- <*> mushes tuples together one way, and takes them apart the other

Invertible Syntax Descriptions: alternative

Alternative[6] is trivial

```haskell
class Alternative where
  (<|>) :: f a -> f a -> f a
```

And we now have an abstract Syntax[7]

```haskell
class (IsoFunctor s, ProductFunctor s, Alternative s)
       => Syntax s where
  pure :: Eq a => a -> s a
```

Invertible Syntax Descriptions: the punchline

Parsing

```
parseMany :: Parser a -> Parser (List a)
parseMany p
  =   const Nil <$> text ""
  <|> Cons      <$> p
                <*> parseMany p
```

Printing

```
printMany :: (a -> Doc) -> (List a -> Doc)
printMany p list
  = case list of
    Nil       -> text ""
    Cons x xs -> p x
              <> printMany p xs
```

Invertible Syntax Descriptions: the punchline

Invertible many

```
many :: Syntax s => s a -> s (List a)
many p
  =  nil  <$> pure ()
 <|> cons <$> p <*> many p
```

Invertible Syntax Descriptions: printer syntax

The implementation of Syntax for Printer

```
instance IsoFunctor Printer where
  iso <$> Printer p
    = Printer (\b -> unapply iso b >>= p)

instance ProductFunctor Printer where
  Printer p <*> Printer q
    = Printer (\(x, y) -> liftM2 (++) (p x) (q y))

instance Alternative Printer where
  Printer p <|> Printer q
    = Printer (\s -> mplus (p s) (q s))

instance Syntax Printer where
  pure x
    = Printer (\y -> if x == y then Just "" else N...)
```

Invertible Syntax Descriptions: summary

Invertible Syntax Descriptions: summary

- Partial isos: composable building blocks for munging data

Invertible Syntax Descriptions: summary

- Partial isos: composable building blocks for munging data
- IsoFunctor: to "lift" theses isos into concrete printers or parsers

Invertible Syntax Descriptions: summary

- Partial isos: composable building blocks for munging data
- IsoFunctor: to "lift" theses isos into concrete printers or parsers
- ProductFunctor: to handle multiple fields and recursion via tuples

Invertible Syntax Descriptions: summary

- Partial isos: composable building blocks for munging data
- IsoFunctor: to "lift" theses isos into concrete printers or parsers
- ProductFunctor: to handle multiple fields and recursion via tuples
- Syntax: to glue all these constraints together and add pure

Let's try it on enterprise JSON!

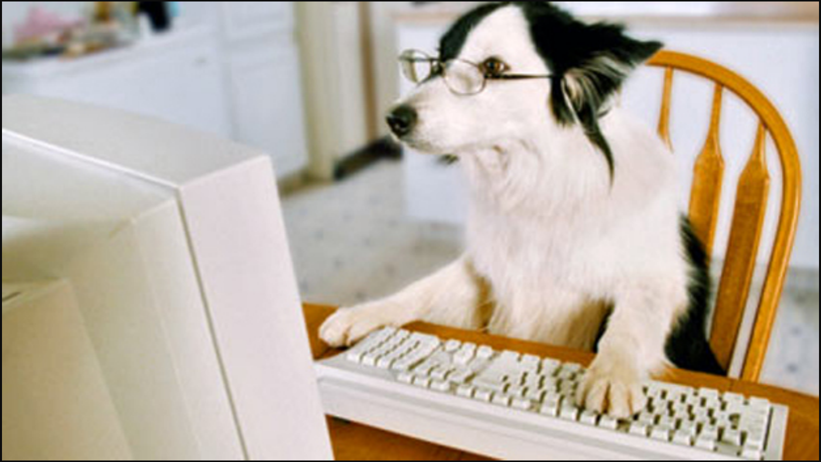


Figure 2: We are now enterprise developers

Let's try it on enterprise JSON!

Two primitives for all your JSON needs:

```
class Syntax s => JsonSyntax s where
    runSub :: s v -> s Value -> s v

    value :: s Value
```

JsonBuilder/Parser IsoFunctor

Starts off simple

```
newtype JsonBuilder a = JsonBuilder
  { runBuilder :: a -> Maybe Value }

newtype JsonParser a = JsonParser
  { runParser :: Value -> Maybe a }

instance IsoFunctor JsonBuilder where
  (<$>) :: Iso a b -> JsonBuilder a -> JsonBuilder b
  i <$> JsonBuilder b = JsonBuilder $ unapply i >=> b

instance IsoFunctor JsonParser where
  (<$>) :: Iso a b -> JsonParser a -> JsonParser b
  i <$> JsonParser p = JsonParser $ apply i <=< p
```

JsonBuilder ProductFunctor

Mush tuples together with applicative when building

```
instance ProductFunctor JsonBuilder where
  (<*>) :: JsonBuilder a
        -> JsonBuilder b
        -> JsonBuilder (a,b)
  JsonBuilder p <*> JsonBuilder q =
    JsonBuilder $ \(a,b) -> do
      a' <- p a
      b' <- q b
      merge a' b'
    where
      merge (Object a) (Object b) =
        Just . Object $ a `union` b
      merge a (Array b) = Just . Array $ V.cons a b
      merge x Null = Just x
      merge Null x = Just x
      merge _ _ = Nothing
```

JsonParser ProductFunctor

Take the things apart and tuple them when parsing

```haskell
instance ProductFunctor JsonParser where
  (<*>) :: JsonParser a -> JsonParser b -> JsonParser (a,b)
  JsonParser p <*> JsonParser q =
    JsonParser $ \v -> do
      let (a,b) | Array x <- v, Just y <- x !? 0
                = (y, Array $ V.tail x)
                | Array _ <- v
                = (Null, Null)
                | otherwise
                = (v,v)
      liftM2 (,) (p a) (q b)
```

JsonBuilder/Parser Alternative

Try one, otherwise the other. Same implementation.

```haskell
instance Alternative JsonBuilder where
  (<||>) :: JsonBuilder a -> JsonBuilder a -> JsonBuilder a
  JsonBuilder p <||> JsonBuilder q =
    JsonBuilder $ \a -> p a `mplus` q a

  empty :: JsonBuilder a
  empty = JsonBuilder $ const Nothing

instance Alternative JsonParser where
  (<||>) :: JsonParser a -> JsonParser a -> JsonParser a
  JsonParser p <||> JsonParser q =
    JsonParser $ \v -> p v `mplus` q v

  empty :: JsonParser a
  empty = JsonParser $ const Nothing
```

JsonBuilder/Parser JsonSyntax

Providing access to underlying JSON Values

```haskell
instance JsonSyntax JsonBuilder where
  value :: JsonBuilder Value
  value = JsonBuilder Just

  runSub :: JsonBuilder v
         -> JsonBuilder Value
         -> JsonBuilder v
  runSub (JsonBuilder a) (JsonBuilder b) =
    JsonBuilder $ a >=> b

instance JsonSyntax JsonParser where
  value = JsonParser Just

  runSub (JsonParser a) (JsonParser b) =
    JsonParser $ a <=< b
```

JsonSyntax combinators

The lens-aeson package provides primitives for most of the combinators we want, but it has these ~~terrifying~~ powerful Prism things:

Prism

```
type Prism s t a b =
    (Choice p, Applicative f) =>
      p a (f b) -> p s (f t)

type Prism' s a = Prism s s a a
```

Review/preview

```
preview :: Prism' a b -> a -> Maybe b
review  :: Prism' a b -> b -> a
```

Prisms/isos are "stronger" than partial isos

Demoting prisms and "real" isos to partial ones

```
demote :: Prism' a b -> Iso a b
demote p = unsafeMakeIso (preview p)
                         (review (_Just . p))
```

JsonSyntax combinators

Given a "free" Prism from lens-aeson

```
_Bool :: Prism' Value Bool
```

JsonSyntax combinators

Given a “free” Prism from lens-aeson

```
_Bool :: Prism' Value Bool
```

We can get an Iso and have a Value

```
demote _Bool :: Iso Value Bool
value :: Syntax s => s Value
```

JsonSyntax combinators

Given a “free” Prism from lens-aeson

```
_Bool :: Prism' Value Bool
```

We can get an Iso and have a Value

```
demote _Bool :: Iso Value Bool
value :: Syntax s => s Value
```

Fmapping these gives us “free” combinators

```
(<$>) :: Iso Value Bool -> s Value -> s Bool

jsonBool :: JsonSyntax s => s Bool
jsonBool = demote _Bool <$> value
```

JsonSyntax combinators

Given a “free” Prism from lens-aeson

```haskell
_Bool :: Prism' Value Bool
```

We can get an Iso and have a Value

```haskell
demote _Bool :: Iso Value Bool
value :: Syntax s => s Value
```

Fmapping these gives us “free” combinators

```haskell
(<$>) :: Iso Value Bool -> s Value -> s Bool

jsonBool :: JsonSyntax s => s Bool
jsonBool = demote _Bool <$> value

jsonNumber :: JsonSyntax s => s Scientific
jsonNumber = demote _Number <$> value

jsonString :: JsonSyntax s => s Text
jsonString = demote _String <$> value
```

JsonSyntax combinators

Looking up keys in objects

```
runSub :: s v -> s Value -> s v

jsonField
    :: JsonSyntax s
    => Text
    -- ^ Key to lookup/insert
    -> s v
    -- ^ Sub-parser
    -> s v
jsonField k syntax = runSub syntax (keyIso <$> value)
  where
    keyIso = demote $ prism' (\x -> Object [(k,x)])
                             (^? key k)
```

JsonSyntax combinators

When you want to ensure something is there

```
is :: (JsonSyntax s, Eq a) => s a -> a -> s ()
is s a = demote (prism' (const a)
                        (guard . (a ==))) <$> s
```

JsonSyntax summary

JsonSyntax summary

- JsonSyntax: to provide access to the underlying domain specific data

JsonSyntax summary

- JsonSyntax: to provide access to the underlying domain specific data
- Prisms are stronger than partial isos

JsonSyntax summary

- JsonSyntax: to provide access to the underlying domain specific data
- Prisms are stronger than partial isos
- lens-aeson made it easy to define JSON combinators

JsonSyntax summary

- JsonSyntax: to provide access to the underlying domain specific data
- Prisms are stronger than partial isos
- lens-aeson made it easy to define JSON combinators
- These combinators can be thought of as relations between abstract syntax and concrete syntax.

Example - Round-tripping balls

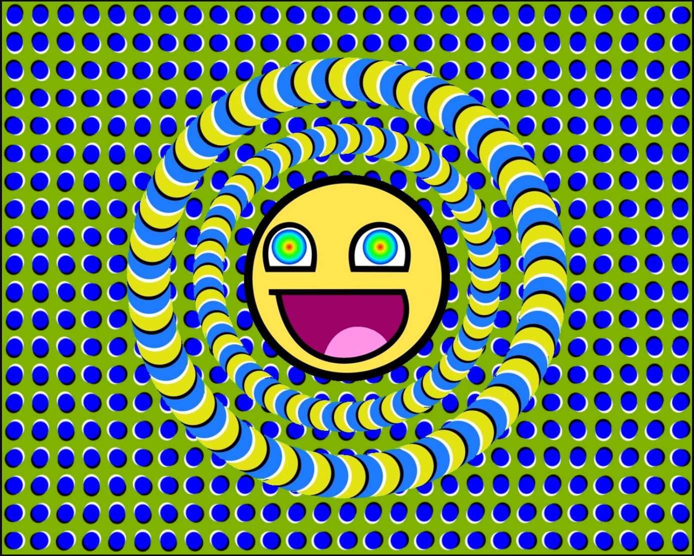


Figure 3: A happy ball

Example - Round-tripping balls

We can have either bouncy or lumpy balls

```
data Ball
    = Lumpy  { _colour      :: Text
             , _lumps       :: [[Bool]]
             }
    | Bouncy { _bouncyness :: Double }
  deriving (Eq, Show)
```

Example - Round-tripping balls

Bouncy balls are happy, lumpy ones are not.

```
[ Lumpy  {_colour = "Rainbow"
         , _lumps = [[True,False],[False,False]]}
, Bouncy {_bouncyness = 3.141592653589793}]
```

⬇

```
[
   {
      "colour" : "Rainbow",
      ":D" : false,
      "lumps" : [[true,false],[false,false]]
   },
   {
      "bouncyness" : 3.14159265358979,
      ":D" : true
   }
]
```

Example - Round-tripping balls

Ball syntax

```
ballSyntax :: JsonSyntax s => s Ball
ballSyntax
  =  lumpy <$> jsonField ":D" (jsonBool `is` False)
            *> jsonField "colour" jsonString
           <*> jsonField "lumps" (many $ many jsonBool)
 <|> bouncy
           <$> jsonField ":D" (jsonBool `is` True)
            *> jsonField "bouncyness" jsonRealFloat
```

Example - Round-tripping balls

The test

```
main :: IO ()
main = do
    let tony   = Lumpy "Rainbow"
                       [[True, False], [False, False]]
    let oscar = Bouncy pi
    let pit    = [tony, oscar]

    let Just blob = runBuilder (many ballSyntax) pit
    L.putStrLn $ encode blob

    let Just pit' = runParser (many ballSyntax) blob
    print pit'
    print $ pit == pit'
```

Example - Round-tripping balls

Output (whitespace added)

```
[  {
       "colour" : "Rainbow",
       ":D" : false,
       "lumps" : [[true,false],[false,false]]
   },
   {
       "bouncyness" : 3.14159265358979,
       ":D" : true
}  ]

[ Lumpy "Rainbow" [[True,False],[False,False]]
, Bouncy 3.141592653589793
]

True
```

Real world example with full library

Currency parser

```
-- | Parse an enterprise currency field, which is a
--   blob of text looking like:
--
--   "00.43"
--
-- This un-/parser retains all precision avaliable.
currency :: JsonSyntax s => s Scientific
currency = demoteLR "currency" (prism' f g) <$> value
  where
    f = String . LT.toStrict . LT.toLazyText . fmt
    g = readMaybe . ST.unpack
      . ST.filter (not . isSpace) <=< preview _String
    -- We render with arbitrary precision (Nothing)
    -- and standard decimal notation (Fixed)
    fmt = LT.formatScientificBuilder Fixed Nothing
```

Real world example with full library

Date time parser

```haskell
-- | Parse an enterprise datetime field, looks like:
--
--    "20/6/2014 4:25 pm"
datetime :: JsonSyntax s => s UTCTime
datetime = demoteLR "datetime" (prism' f g) <$> value
  where
    f = String . ST.pack . opts formatTime
    g = opts parseTime . ST.unpack <=< preview _String
    opts h =
      h defaultTimeLocale "%-d/%-m/%Y %-l:%M %P"
```

Summary/Conclusion

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.
- Problem: writing round-trip printer/parsers with the get/put idiom is redundant and error prone.

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.
- Problem: writing round-trip printer/parsers with the get/put idiom is redundant and error prone.
- Generics/TH too rigid, libraries too flexible.

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.
- Problem: writing round-trip printer/parsers with the get/put idiom is redundant and error prone.
- Generics/TH too rigid, libraries too flexible.
- A reasonable middle ground is detailed in the paper by Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing.

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.
- Problem: writing round-trip printer/parsers with the get/put idiom is redundant and error prone.
- Generics/TH too rigid, libraries too flexible.
- A reasonable middle ground is detailed in the paper by Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing.
- I'd love to work with people on improving the current machinery.

Summary/Conclusion

- Invertible syntax descriptions are a way to write better printer/parsers such that we type less, think less and make fewer mistakes.
- Problem: writing round-trip printer/parsers with the get/put idiom is redundant and error prone.
- Generics/TH too rigid, libraries too flexible.
- A reasonable middle ground is detailed in the paper by Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing.
- I'd love to work with people on improving the current machinery.
- Our library: http://github.com/anchor/roundtrip-aeson

A note on categories

Ekmett's "categories" package, more category-like functors.

```
import qualified Control.Categorical.Functor as CF

type Hask = (->)

instance CF.Functor JsonBuilder Iso Hask where
    fmap iso (JsonBuilder f) =
        JsonBuilder (unapply iso >=> f)
```