Unifying parsing and prettyprinting

Sergey Vinokurov

Outline

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don't repeat yourself

Why parse and prettyprint are useful together?

- Programming languages and external DSLs
- Structured data - can "prettyprint" to a tree-like structure. E.g.
  - JSON
  - XML
- Serialization/deserialization
- The problem: *must keep them in sync*
  - DRY - don’t repeat yourself

Our example

- We'll use simple expression language to drive the discussion
- Arithmetic expressions with literals, addition and multiplication

```
data Expr =
    Lit Int
  | Add Expr Expr
  | Mul Expr Expr
  deriving (Show)
```

Sample expression parser

```
exprParser :: Parser Expr
exprParser = pAdd

pAdd = pMul <|>
       Add <$> pMul <* pChar '+' <*> pAdd

pMul = pAtomic <|>
       Mul <$> pAtomic <* pChar '*' <*> pMul

pAtomic =
  Lit <$> pInt <|>
  bracket (pChar '(') (pChar ')') exprParser
```

Parsing

- Goging from string to a tree-like structure
- May fail if input is invalid
- It’s a Covariant Functor - a producer of values
- Many parsing combinator library support at least Applicative interface I.e. they share some standard set of combinators

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

Prettyprinting

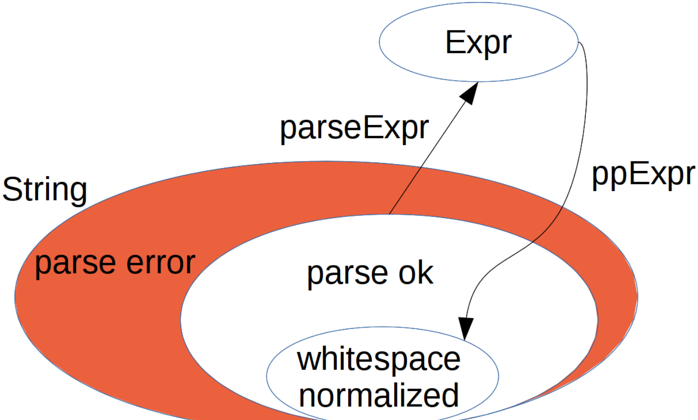
- Going from a tree-like structure to a string
- Usually does not fail - can always produce a string given some Expr
- However, we'll need to support a notion of failure
- It's a Contravariant Functor - a consumer of values
- Usually there's no standard set of combinators that prettyprinting libraries support
- Most of the time the interface is somewhat different than for parsers - a typeclass for values, that can be prettyprinted

```
newtype Printer a = Printer (a -> Maybe String)
```

Relationship between parsing and prettyprinting

Parsing and prettyprinting are almost inverses of one another.

```
parseExpr :: String -> Either String a
ppExpr    :: a -> String
```



Parsing/prettyprinting laws

Well-behaved prettyprinting should produce a string that results in the original expression, when parsed.

$$\text{parseExpr} \circ \text{ppExpr} = \text{id}$$

Parsing/prettyprinting laws

Well-behaved prettyprinting should produce a string that results in the original expression, when parsed.

$$\text{parseExpr} \circ \text{ppExpr} = \text{id}$$

However, for ($\text{ppExpr} \circ \text{parseExpr}$) this is not the case

Parsing/prettyprinting laws, continued

- After single cycle of parsing and prettyprinting the string whitespace normalizes.
- Code formatters work this way
- Formatting a second time does not change anything
- $(\text{ppExpr} \circ \text{parseExpr})$ is idempotent, $f(f\ x) = f\ x$

$$\text{ppExpr} \circ \text{parseExpr} = \\ (\text{ppExpr} \circ \text{parseExpr}) \circ (\text{ppExpr} \circ \text{parseExpr})$$

Building syntax description combinators

The basic things we're operating on are characters. We can parse current character - get one from input, if we're not at eof.
We can add given character to our pretty output.

```
getChar :: Parser Char
ppChar  :: Printer Char
```

Let's call this bit a *token*. It's a basic syntax description, *s*, that works with characters.

```
token :: s Char
```

Semantic actions

- Want to get `s a` out of `s Char`
- Need Functor interface for syntax descriptions
- Must provide means to parse `a` from string as well as prettyprint it to string at the same time

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

fmapParser :: (a -> b) -> Parser a -> Parser b
fmapParser f (Parser g) =
  Parser $ map (\(x, str) -> (f x, str)) . g
```

Semantic actions for Printer

- The prettyprinter `Printer a` is a, so called, Contravariant functor
- It consumes values of type a and produces string
- There's no vanilla Functor instance for it

```
-- Trying to write vanilla functor instance.
f :: (a -> b) -> Printer a -> Printer b

-- Expand Printer definition.
-- Cannot write this function.
f :: (a -> b) -> (a -> String) -> (b -> String)
```

Partial isomorphisms

- Functor or Contravariant alone are not enough
- They allow to go in only one direction, syntax description must support both
- Use partial invertible functions that allow to go in both directions

Partial isomorphisms, continued

- Partiality is useful here as we don't want to confine ourselves to restrictive universe of total invertible functions

```
data Iso a b =
  Iso (a -> Maybe b) (b -> Maybe a)

apply :: Iso a b -> a -> Maybe b
apply (Iso f _) = f

unapply :: Iso a b -> b -> Maybe a
unapply (Iso _ g) = g
```

Partial isomorphisms, continued

- Partiality is useful here as we don't want to confine ourselves to restrictive universe of total invertible functions

```
data Iso a b =
  Iso (a -> Maybe b) (b -> Maybe a)

apply :: Iso a b -> a -> Maybe b
apply (Iso f _) = f

unapply :: Iso a b -> b -> Maybe a
unapply (Iso _ g) = g
```

$\forall x, y : \text{apply iso } x = \text{Just } y \iff \text{unapply iso } y = \text{Just } x$

IsoFunctor

- Define our own Functor-like class
- If isomorphism fails, our Parser and Printer will fail too

```
class IsoFunctor f where
  (<$$>) :: Iso a b -> f a -> f b
infixr 4 <$$>

instance IsoFunctor Parser where
  iso <$$> Parser p = Parser $ \s ->
    [ (y, s')
    | (x, s') <- p s
    , Just y  <- [apply iso x]
    ]

instance IsoFunctor Printer where
  iso <$$> Printer g = Printer $
    unapply iso >=> g -- Maybe monad
```

Parsing sequences

- Need a way to express "parse X followed by Y"
- Will use Applicative-like interface
  - Less powerful than monads
  - Provides just enough power to parse context-free grammars

Applicative

- The Applicative class is designed for covariant functors - producers of values
- As with Functor, cannot implement this interface for Printer
- Reformulation of Applicative - ProductFunctor

```
class (IsoFunctor f) => ProductFunctor f where
  (<**>) :: f a -> f b -> f (a, b)

infixr 5 <**>

instance ProductFunctor Parser where
  Parser p <**> Parser q = Parser $ \s ->
    [ ((x, y), s'')
    | (x, s')  <- p s
    , (y, s'') <- q s'
    ]
```

Printer instance

```
class (IsoFunctor f) => ProductFunctor f where
  (<**>) :: f a -> f b -> f (a, b)

instance ProductFunctor Printer where
  Printer p <**> Printer q = Printer $
    \(x, y) -> liftA2 (++) (p x) (q y)

liftA2
  :: (Applicative f)
  => (a -> b -> c) -> f a -> f b -> f c
```

The final bit: Alternative

- This time need to support a notion "parse X or parse Y if parsing X fails"
- There's starndard class for this called Alternative, but it depends on Applicative
- Define alternative Alternative called PureAlternative!

```
class PureAlternative f where
  -- parser or printer that always fails
  emptyAlt :: f a
  (<||>)   :: f a -> f a -> f a

infixl 3 <||>
```

Alternative instances

```
instance PureAlternative Parser where
  Parser p <||> Parser q = Parser $ \s -> p s +
  emptyAlt               = Parser $ const []

instance PureAlternative Printer where
  Printer p <||> Printer q = Printer $ \x ->
    p x <|> q x
  emptyAlt                 = Printer $ \_ -> No
```

Putting it all together

```
class ( IsoFunctor s
      , ProductFunctor s
      , PureAlternative s
      ) => Syntax s where
  token :: s Char
  -- Eq constraint is for printer
  pureSyn :: (Eq a) => a -> s a
```

Syntax for Parser

```
instance Syntax Parser where
  pureSyn x = Parser $ \s -> [(x, s)]
  token = Parser f
    where
      f (c:cs) = [(c, cs)]
      f []     = []
```

Syntax for Printer

```
instance Syntax Printer where
  pureSyn x = Printer $ \x' ->
    if x == x'
    then Just []
    else Nothing
  token  = Printer $ \c -> Just [c]
```

Parsing digits

```
subset :: (a -> Bool) -> Iso a a
subset p = Iso f f
  where
    f x | p x        = Just x
        | otherwise = Nothing

digit :: (Syntax s) => s Char
digit = subset isDigit <$$> token

isDigit :: Char -> Bool
isDigit c = '0' <= c && c <= '9'
```

Utilities for parsing sequences

```
isoNil :: Iso () [a]
isoNil = Iso f g
  where
    f () = Just []
    g [] = Just ()
    g _  = Nothing

isoCons :: Iso (a, [a]) [a]
isoCons = Iso f g
  where
    f (x, xs) = Just $ x : xs
    g (x:xs)  = Just (x, xs)
    g []      = Nothing
```

Utilities for parsing sequences, continued

```
pmany :: (Syntax s) => s a -> s [a]
pmany p = isoNil <$$> pureSyn () <||>
          isoCons <$$> p <**> pmany p

pmany1 :: (Syntax s) => s a -> s [a]
pmany1 p = isoCons <$$> p <**> pmany p
```

Parsing numbers

```
inverse :: Iso a b -> Iso b a
inverse (Iso f g) = Iso g f

decimal :: Iso Int String
decimal = Iso f g
  where
    f = Just . show
    g str | all isDigit str
          = Just $
            foldl' (\a x -> a * 10 + h x) 0 str
          | otherwise
          = Nothing
    h x = ord x - ord '0'

integer :: (Syntax s) => s Int
integer = inverse decimal <$$> pmany digit
```

Utilities for parsing expressions

Can derive these via Template Haskell

```
lit :: Iso Int Expr
lit = Iso f g
  where
    f n = Just $ Lit n
    g (Lit n) = Just n
    g _       = Nothing
```

Utilities for parsing expressions, continued

```
add :: Iso (Expr, Expr) Expr
add = Iso f g
  where
    f (x, y)      = Just $ Add x y
    g (Add x y) = Just (x, y)
    g _           = Nothing

mul :: Iso (Expr, Expr) Expr
mul = Iso f g
  where
    f (x, y)      = Just $ Mul x y
    g (Mul x y) = Just (x, y)
    g _           = Nothing
```

More utilities for parsing expressions

```
(**>) :: (Syntax s) => Char -> s a -> s a
(**>) c s = Iso f g <$$> token <**> s
  where
    f (c′, x) | c == c′    = Just x
              | otherwise = Nothing
    g x = Just (c, x)

between
  :: (Syntax s) => Char -> Char -> s a -> s a
between l r s =
  Iso f g <$$> token <**> s <**> token
  where
    f (l′, (x, r′))
      | l == l′ && r == r′ = Just x
      | otherwise          = Nothing
    g x = Just (l, (x, r))
```

Parsing expressions

```
expr :: (Syntax s) => s Expr
expr = add <$$> factor <**> '+' **> expr <||>
       factor

factor :: (Syntax s) => s Expr
factor = mul <$$> atomic <**> '*' **> factor <|
         atomic

atomic :: (Syntax s) => s Expr
atomic = lit <$$> integer <||> between '(' ')'
```

Test run

```
runParser :: Parser a -> String -> Maybe a
runParser (Parser p) str =
  case dropWhile (not . null . snd) $ p str of
    (x, []):_ -> Just x
    _         -> Nothing

runPrinter :: Printer a -> a -> Maybe String
runPrinter (Printer p) = p

> runParser expr "10*(2+3)"
Just (Mul (Lit 10) (Add (Lit 2) (Lit 3)))

> runParser expr "(10)*((2)+(3))" >>=
    runPrinter expr
Just "10*(2+3)"
```

Questions?

Questions?

PS btw, we are hiring