# A Monadic Framework for Bidirectional Programming

## Motivation

- Two ways to run a program: avoids code duplication.
- Monads have expressiveness and readability: they might make bidirectional programming easier.

## Three types of bidir. transformations (BX)

- An *invertible parser* can be turned around into a printer.

```
data InvParser x a = InvParser
  { parse :: String -> (String, a)
  , print :: x -> (String, a) }
```

- A *lens* lifts functions (a -> x) into updates on s.

```
data Lens s x a = Lens
  { get :: s -> a
  , set :: x -> s -> (s, a) }
```

- A *generable set* consists of a random generator and a membership function.

```
data GSet x a = GSet
  { generate :: Gen (Maybe a)
  , predicate :: x -> Maybe a }
```

## Monadic bidirectional transformations

- A pair of **reader** and **writer** transformations.
- Parameterized by an *input* type x and an *output* type a.

```
data MBX r w x a = MBX
  { reader :: r a
  , writer :: x -> w a }
instance (Monad r, Monad w) => Monad (MBX r w x)
```

## Common interpretation

- Readers map a **source** s to some **view** a.
- Writers take a value x containing a view a, and instantiate a **partially defined source** $s^!$ such that reading from any complete instantiation yields back a.

  A printer "writes" a string by incrementally instantiating an unknown prefix (_). The final string is obtained by substituting the last unknown with the empty string "".

```
(_) > ("1" ++ _) > ("1 0" ++ _) > ("1 0 -" ++ _)
    > ... > ("1 0 - - 2 - 3 - -" ++ _)
```

- We expect that every written view can be read back.

  This is analogous to the *PutGet* lens law.

## References


- *Combinators for bidirectional tree transformations: A linguistic approach to the view update problem.* Foster et al. (POPL'05)
- *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing.* Rendel and Ostermann. (Haskell'10)
- *Applicative bidirectional programming with lenses.* Matsuda and Wang. (ICFP'15)
- *Beginner's Luck: A Language for Property-Based Generators.* Lampropoulos et al. (POPL'17)


## Monadic profunctors (MP)

- MBXs are **monadic profunctors**.
- A monadic profunctor is a *monad*,

```
return :: a -> p x a
(>>=) :: p x a -> (a -> p x b) -> p x b
```

- also a *contravariant functor* (*cofunctor*),

```
(=.) :: (y -> x) -> p x a -> p y a
```

- such that (f =.) is a *monad morphism* for all f.

```
> f =. return a     =     return a
> f =. (  ma  >>= (\a ->      amb a))
  = (f =. ma) >>= (\a -> f =. amb a)
```

## Example: Parsing and printing trees

In the definition of a MBX like tree, every action is annotated with its "location" in the final result. Erasing these annotations in gray below reveals the code of a parser.

```
word :: InvParser String String
data Tree = L | Node Integer Tree Tree

tree :: InvParser Tree Tree
tree = do
  w <- firstWord =. word
  case w of
    "-" -> return L
    _   -> do l <- nodeLeft  =. tree
              r <- nodeRight =. tree
              return (Node (read w) l r)
  where firstWord L = "-"
        firstWord (Node v _ _) = show v

example = Node 1 (Node 0 L L)
                 (Node 2 L (Node 3 L L))
output  = "1 0 - - 2 - 3 - -"

> runPrinter (print tree example) = output
> runParser  (parse tree) output  = example
```

A more realistic example of invertible parser: https://github.com/Lysxia/unparse-attoparsec.

## BX xor MP

- BX but not MP: bijections.
- MP but not BX: "Profunctor HOAS". (https://www.schoolofhaskell.com/user/edwardk/phoas)

## Future directions

- Refine MPs, e.g., as cofunctors in a category of arrows.
- Interpretation still needs adjustment for GSet.
- How to derive reader from writer (or conversely).


**Li-yao Xia** **Meng Wang** (supervisor)

li-yao.xia@ens.fr, M.W.Wang@kent.ac.uk École Normale Supérieure and University of Kent