A Monadic Framework for Bidirectional Programming

Li-yao Xia (École Normale Supérieure)

Programming with data under different representations is a commonplace problem. The relationship between such representations is most directly expressed using pairs of mappings, called *bidirectional transformations*: a function and its inverse; parser and printer; getter and setter. These transformations are often expected to satisfy consistency properties, and missing them is a frequent source of bugs.

To ease the programming of bidirectional transformations, various frameworks have been proposed. A general approach many of these proposals take is the design of a set of *combinators* which guarantee the good behavior of composite transformations. [3–7]

Designers naturally strive for types and combinators which are similar to or match common abstractions. In particular, monads are known to model various sorts of computations, and syntactic support such as Haskell's do-notation gives monads a rather ergonomic interface.

However, at first glance, it does not seem possible to adapt monads and similar concepts to bidirectional transformations: bidirectionality typically implies invariance, whereas monads must be covariant functors. A common workaround is to redefine existing concepts in a bidirectional setting. In Haskell for instance, Rendel and Ostermann [7] replace the `Functor` typeclass, actually representing functors from a category of functions, with functors from some category of isomorphisms.

By contrast, I propose a way to work directly with existing abstractions by modelling bidirectional computations as *monadic profunctors*: types `p` with two parameters, which are contravariant functors on the first and monads on the second, i.e., we can implement the following signatures:

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

In Haskell, `return` and `(>>=)` belong to the `Monad` typeclass, from the standard library `base`. The operator `(=.)` corresponds to `lmap` from the `Profunctor` typeclass, which is extensively used by the lens library in particular.

Below is a particularly interesting type, which combines two unidirectional computations into a bidirectional one.

```
data Codec g p x a = Codec
  { get :: g a
  , put :: x -> p a }

instance (Functor g, Functor p)
        => Profunctor (Codec g p)
instance (Monad g, Monad p)
        => Monad (Codec g p x)
```

It is a variation of a type found in the codec library in Haskell, which we borrowed the name of[2]. The occurrence of `a` in the `put` component turns out to be crucial for this type to be a `Monad` and thus for the expressiveness of `Codec`.

For example, we can use it to represent a composable pair of parser and printer:

```
type PPCodec x a = Codec Parser Printer x a
data Parser a = Parser (String -> (String, a))
data Printer a = Printer (String, a)
```

Given a primitive to parse/print tokens,

```
data Token = Op Char | Lit Int

token :: PPCodec Token Token
```

we may define the following parser/printer for expressions of binary operations in prefix notation. It takes the shape of a monadic parser, and `(=.)` provides complementary mappings for the printer which can thus use the same definition.

```
data Exp = BinOp Char Exp Exp | Val Int

exp :: PPCodec Exp Exp
exp = do
  t <- leadToken =. token
  case t of
    Op c -> BinOp c
      <$> (operand1 =. exp)
      <*> (operand2 =. exp)
    Lit n -> return (Val n)

leadToken (BinOp c _ _) = Op c
leadToken (Val n) = Int n

operand1 (BinOp _ e _) = e
operand2 (BinOp _ _ e) = e
```

Lenses are a general kind of bidirectional transformation we may also represent as a `Codec`. They are traditionally defined with the following type:

```
data Lens s v = Lens
  { get :: s -> v
  , put :: v -> s -> s }
```

After separating the positive and negative positions of the variable `v`, we obtain `Codec (Reader s) (State s)`. Below is a simplified equivalent type:

```
data CodecLens s v w = CodecLens
  { get :: s -> w
  , put :: v -> s -> (s, w) }
```

We may compose such lenses monadically like we did earlier for printers/parsers. This also extends a previous proposal to program lenses in applicative style [5].

In addition to parsers/printers and lenses, the abstractness of monadic profunctors suggests that they may fit well with various other bidirectional frameworks.

I am investigating the practicability of programming with monadic profunctors. Early experimentation finds that they enable the composition of bidirectional transformations in rich and general ways from domain-specific primitives.

[1] Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J. and Stevens, P. 2015. Notions of bidirectional computation and entangled state monads. *Mathematics of program construction* (June 2015).

[2] Chilton, P. 2015. codec: First-class record construction and bidirectional serialization. http://hackage.haskell.org/package/codec.

[3] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C. and Schmitt, A. 2005. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *Proceedings of the 32Nd acm sigplan-sigact symposium on principles of programming languages* (New York, NY, USA, 2005), 233–246.

[4] Kennedy, A.J. 2004. FUNCTIONAL pearl pickler combinators. *J. Funct. Program.* 14, 6 (Nov. 2004), 727–739.

[5] Matsuda, K. and Wang, M. 2015. Applicative bidirectional programming with lenses. *Proceedings of the 20th acm sigplan international conference on functional programming* (New York, NY, USA, 2015), 62–74.

[6] Pacheco, H., Hu, Z. and Fischer, S. 2014. Monadic combinators for "putback" style bidirectional programming. *Proceedings of the acm sigplan 2014 workshop on partial evaluation and program manipulation* (New York, NY, USA, 2014), 39–50.

[7] Rendel, T. and Ostermann, K. 2010. Invertible syntax descriptions: Unifying parsing and pretty printing. *Proceedings of the third acm haskell symposium on haskell* (New York, NY, USA, 2010), 1–12.