Monadic Bidirectional Programming

Li-yao Xia[1], Dominic Orchard[2], and Meng Wang[2]

[1] University of Pennsylvania
[2] University of Kent

Abstract. Software frequently converts data from one representation to another and vice versa. Instead of naively specifying both directions separately, which is error prone and introduces duplication, various *bidirectional programming* techniques exist to develop programs which may be interpreted in both directions. However, these techniques often employ programming idioms that are alien to non-experts.
We propose a framework for composing bidirectional programs monadically, reusing familiar abstractions in functional programming languages such as Haskell. We demonstrate the expressiveness of our approach with applications to writing parsers/printers, lenses, and also generators/predicates. Finally, we show how to leverage compositionality and equational reasoning for the verification of *round-tripping properties* for such monadic bidirectional programs.

1 Introduction

A *bidirectional transformation* (BX) is a pair of mutually related mappings between source and target data objects. This pattern is found in a wide range of applications, such as parsing (Rendel and Ostermann, 2010), databases (Bancilhon and Spyratos, 1981), XML transformation (Pacheco et al., 2014b), etc.

Tediously, one can separately construct the corresponding forwards and backwards functions of a BX. This approach duplicates effort, is prone to error, and causes subsequent maintenance issues. This can be avoided using specialised programming languages that generate functions in both directions from a single definition (Foster et al., 2007; Voigtländer, 2009; Matsuda et al., 2007), a discipline known as *bidirectional programming*.

Lenses The most well known language family for BX are *lenses* (Foster et al., 2007). A lens captures transformations between *sources* S and *views* V via a pair of functions $\texttt{get} : S \to V$ and $\texttt{put} : S \to V \to S$. The `get` function extracts a view from a source and `put` takes an updated view and a source as inputs to produce an updated source. The asymmetrical nature of `get` and `put` provides a way to compensate for the forgetful nature of programs, making it possible to recover some of the source data that is not present in the view. In other words, `get` does not have to be injective to have a corresponding `put`.

Bidirectional transformations typically respect *round-tripping* laws, capturing the extent to which the transformations preserve information between the

two data representations. For example, *well-behaved lenses* (Bohannon et al., 2006; Foster et al., 2007) should satisfy the following laws:

$$\mathsf{put}\ s\ (\mathsf{get}\ s) = s \qquad\qquad \mathsf{get}\ (\mathsf{put}\ s\ v) = v$$

Lens languages are conventionally designed to make invariants of these properties. This focus on unconditional correctness inevitably leads to reduced practicality in programming. Lens combinators are typically stylised and disconnected from established programming idioms. For example, they do not benefit from the applicative/monad framework which combinator libraries typically exploit in other domains. Moreover, it appears that many applications of bidirectional transformations (for example, parsers and printers) do not share the lens *get*/*put* pattern, and as a result are out of reach.

Contributions In this paper, we deliberately avoid this well-tried approach, exploring instead a novel point in the BX design space based on monadic programming, naturally reusing host language constructs. We revisit lenses, and two more bidirectional patterns, and demonstrate how they can be subject to bidirectional programming. By being uncompromising about the monad interface, we expose the essential ideas behind our framework whilst maximizing its practical usability. The trade off of this design is that we can no longer perform correctness reasoning in the same way as conventional lenses: our interface does not rule out all non-round-tripping BX. We tackle this issue by proposing a new compositional reasoning framework that is flexible enough to characterise a variety of round-tripping properties. Specifically, we make the following contributions:

- We describe a method to enable *monadic composition* for bidirectional programs.
- To demonstrate the flexibility of our approach, we apply the above method to three different problem domains: parsers/printers, lenses, and generators/predicates for structured data. While the first two are well-explored areas in the bidirectional programming literature, we know of little work on the third one.
- We present a scalable reasoning framework, capturing notions of *compositionality* for bidirectional properties. We define classes of round-tripping properties inherent to bidirectionalism, which can be verified by following simple criteria. These principles are demonstrated with our three examples.
- We have implemented these ideas as Haskell libraries[3], with two wrappers around attoparsec for parsers and printers, and QuickCheck for generators and predicates, showing the practicality of our approach.

Throughout we use Haskell for concrete programming language examples, but the programming patterns can be easily expressed in many functional languages. We will use the Haskell notation of assigning type signatures to expressions via the infix double colon "::".

[3] `https://github.com/Lysxia/profunctor-monad`

1.1 Background

We introduced lenses briefly above. In this subsection, we introduce the other two bidirectional examples used throughout this paper: *parsers/printers* and *generators/predicates*.

Parsing and printing Programming language tools (such as interpreters, compilers, and refactoring tools) typically require two intimately linked components: *parsers* and *printers*, respectively mapping from source code to ASTs and back. A simple implementation of these two functions has the types:

```
parser :: String → AST          printer :: AST → String
```

Parsers and printers are rarely actual inverses to each other, but instead typically exhibit a variant of round-tripping such as:

```
parser ∘ printer ∘ parser ≡ parser    printer ∘ parser ∘ printer ≡ printer
```

The left equation describes the common situation that parsing discards information about source code, such as whitespace, so that printing the resulting AST does not recover the original source. However, printing retains enough information such that parsing the printed output yields an AST which is equivalent to the AST from parsing the original source. The right equation describes the dual: printing may map different ASTs to the same string. For example, printed code $1 + 2 + 3$ might be produced by left- and right-associated syntax trees.

For particular AST subsets, printing and parsing may be left- or right- inverses to each other, but the above two laws capture the wider pattern of interaction seen between printers and parsers. Other characterisations are also possible, with equivalence classes of ASTs (accounting for reassociations) or of source code strings (that account for whitespace and syntactic sugar), providing inverses.

Alternatively, parsers and printers may satisfy properties about the interaction of partially-parsed inputs with the printer and parser, *e.g.*:

```
(let (x, s') = parser s in parser ((printer x) ++ s'))  ≡  parser s
```

Thus, parsing and printing follows a pattern of inverse-like functions which does not fit the lens paradigm. The pattern resembles lenses between a source (source code) and view (ASTs), but with a compositional notion for the source and partial "gets" which consume some of the source, leaving a remainder.

Writing parsers and printers by hand is often tedious due to the redundancy implied by that inverse-like relation. Thus, various approaches have previously been proposed specifically for reducing the effort of developing parsers/printers, by generating both from a common definition Rendel and Ostermann (2010); Matsuda and Wang (2013).

Generating and checking Property-based testing (*e.g.*, QuickCheck) expresses program properties as executable predicates. For instance, the following property checks that an insertion function `insert`, given a sorted list — as checked

by the predicate `isSorted :: [Int] → Bool` — produces another sorted list. The combinator $\Longrightarrow$ represents implication for properties.

```
propInsert :: Int → [Int] → Property
propInsert val list = isSorted list ⟹ isSorted (insert val list)
```

To test it, a testing framework generates random inputs `val` and `list`. It first checks whether `list` is sorted, and if it is, checks that `insert val list` is sorted as well; this process is repeated until either a counterexample is found or a predetermined number of test cases pass.

However, this naïve method is inefficient: many properties such as `propInsert` have preconditions which are satisfied by an extremely small fraction of inputs. In this case, the ratio of sorted lists among lists of length n is inversely proportional to $n!$, so most generated inputs will be discarded for not satisfying the `isSorted` precondition. Such tests give no information about the validity of the predicate being tested and thus are prohibitively inefficient.

When too many inputs are being discarded, the user must instead supply the framework with *custom generators* of values satisfying the precondition: `genSorted :: Gen [Int]`.

One can expect two complementary properties of such a generator. A generator is *sound* with respect to the predicate `isSorted` if it generates only values satisfying `isSorted`; soundness means that no tests are discarded, hence the property to test is better exercised. A generator is *complete* with respect to `isSorted` if it can generate all satisfying values; completeness ensures the correctness of testing a property with `isSorted` as a precondition, in the sense that if there is a counterexample, it will be generated eventually. In this setting of testing, completeness, which affects the potential adequacy of testing, is arguably more important than soundness, which affects only efficiency.

It is clear that generators and predicates are closely related, forming a pattern similar to that of bidirectional transformations. Given that good generators are usually difficult to construct, being able to extracting both from a common specification with bidirectional programming is a very attractive alternative.

Roadmap We begin by outlining the core of our approach in Section 2: using *monadic profunctors* to structure bidirectional programs. This section uses parsers and printers as a concrete example, whilst also explaining the general approach. Section 3 then presents a compositional reasoning framework for monadic bidirectional programs, with varying degrees of strength adapted to different round-tripping laws. Then, we replay the developments of the previous sections to define lens as well as generators and predicates in Sections 4 and 5.

2 Monadic bidirectional programming

In this section, we show how to write bidirectional parsers and printers which can be composed monadically.

Notation The variables `x, y, z` will have types `u, v, w`, respectively. Let `m` be some monad. With a similar convention for the second letter, the variable `py` will represent monadic actions, of type `m v`, `kz` will represent monadic continuations, or Kleisli arrows, of type `v → m w`.

Parsers as monads Let us use the simple type for parsers from the introduction, but abstracted on the AST type:

```
data Parser v = Parser { parse :: String → (v, String) }
```

It is well know that parsers are monadic (Hutton and Meijer, 1998), *i.e.*, they have a well-defined notion of sequential composition (Wadler, 1995), embodied by the following interface:

```
instance Monad Parser where
  return :: v → Parser v
  (>>=) :: Parser v → (v → Parser w) → Parser w
```

This interface has the following implementation which we will briefly describe:

```
  return y  = Parser (λs → (y, s))
  py >>= kz = Parser (λs → let (y, s') = parse py s in parse (kz y) s')
```

For any value `y`, the parser `return y` doesn't consume any input and its result is `y`. The sequential composition operator `(>>=)`, called *bind*, first runs the parser `py`, resulting in a value `y` which is used to create the parser `kz y`, which is in turn run on the remaining input `s'`.

Making printers monadic Are printers also monadic? Printers have the type:

```
data Printer v = Printer { print :: v → String }
```

This type cannot be a monad, as monads must be covariant functors but `Printer` is *contravariant*. The type parameter `v` corresponds to the *output* of `Parser v`, whereas it corresponds to the *input* of `Printer v`.

Instead, we modify the type of printers to have an extra type parameter which is an output:

```
data Printer u v = Printer { print :: u → (String, v) }
```

The input type `u` can be understood as some broader representation of a syntax tree with a certain subtree of type `v`. A printer extracts that subtree from the input, and returns it alongside its encoding as a string.

For any fixed input type, that new type of printer is indeed monadic.

```
instance Monad (Printer u) where
  return :: v → Printer u v
  return y = Printer (λ_ → ("", y))

  (>>=) :: Printer u v → (v → Printer u w) → Printer u w
  py >>= kz = Printer (λx →
              let (s, y) = print py x
                  (s', z) = print (kz y) x in (s ++ s', z))
```

The printer `return y` ignores its input and prints nothing. For sequential composition, an input `x` is shared by both computations and the resulting strings are concatenated. Note that it would not be possible to define `(>>=)` if the input and output types were always the same.

2.1 Biparsers

A parser and a printer together make a *biparser* (*bidirectional parser*):

```
data P u v = P { parse :: String → (v, String)
               , print :: u      → (String, v) }
```

Thus, we can read `P u v` as the type of a parser into `v` values and a printer of `v` values derived from `u` values. When the types of inputs `u` and outputs `v` are the same we are in a more familiar territory, where the printer may simply return its input alongside the printed string.

The definition of `P` allows both parsers and printers to be composed sequentially at the same time. That is, `P u` is a monad for any type `u`:

```
instance Monad (P u) where
  return :: v → P u v
  return y = P (λ s → (y, s)) (λ _ → ("", y))

  (>>=) :: P u v → (v → P u w) → P u w
  py >>= kz = P parser printer where
    parser  s = let (y, s1) = parse py s in parse (kz y) s1,
    printer x = let (s0, y) = print py x
                    (s1, z) = print (kz y) x in (s0 ++ s1, z)
```

The type of `(>>=)` restricts the input type `u` of both operands (`py` and `kz`) and the result to be the same. We can then define the following function `comap` to transform the input type of the printer, leaving the parser untouched:

```
comap :: (u → u') → P u' v → P u v
comap f (P parse print) = P parse (print ∘ f)
```

An example Let us perform the following task: write a biparser for strings which are prefixed by their length and a space, `string :: P String String`.

For example, the following unit tests should be true:

```
test1 = parse string "6␣lambda␣calculus" == ("lambda", "␣calculus")
test2 = print string "SKI" == ("3␣SKI", "SKI")
```

We start by defining a primitive bi-parser of single characters as:

```
char :: P Char Char
char = P (λ (c : s) → (c, s)) (λ c → ([c], c))
```

A character is parsed by deconstructing the source string into its head and tail. For brevity, we do not handle the failure associated with an empty string. A character `c` is printed as its single-letter string paired with `c`.

Next, we define a biparser `digit` for an integer followed by a single space. In Haskell, the **do**-notation statement "`d ← comap head char`" desugars to use `(>>=)` and a function: "`comap head char >>= λ d →`".

```
digits :: P String String
digits = do
  d ← comap head char
  if isDigit d then do
    igits ← comap tail digits
    return (d : igits)
  else if d == ' ' then
    return "␣"
  else
    error "Expected␣digit␣or␣space."
```

```
int :: P Int Int
int = do
  ds ← comap show' digits
  return (read ds)
  where
    show' n = show n ++ "␣"
```

(A safer implementation could return the `Maybe` type when parsing but we keep things simple here.) On the left, `digits` extracts a `String` consisting of digits followed by a single space. As a parser, it parses a character (`comap head char`), if it is a digit then it continues parsing recursively (`comap tail digits`), and appends the first character to the result (`fmap (c :)`), otherwise, it must be a space, in which case the parser terminates by returning it. As a printer, it expects a string of the same format, which must be non-empty; it writes the first character through `comap head char`, which returns it as `c`; if it is a digit, then there must be more to print since the last character should be a space, otherwise the printer terminates.

On the right, the biparser `int` uses `fmap read` to convert an input string of digits parsed by `digits` to an integer, and `comap show'` to convert an integer to an output string printed by `digits`,

After parsing an integer `n`, we can parse the string following it by iterating `n` times the biparser `char`. This is captured by the `replicateP` combinator below, akin to `replicateM` from Haskell's standard library. It is defined recursively like `digits`, but the termination condition is given by an external parameter. To iterate `n` times a parser `py`: if `n == 0`, there is nothing to do and we return the empty list; otherwise for `n > 0`, we run `py` once to get the head `y`, and recursively iterate `n-1` times to get the tail `ys`.

Note that although not apparent in its type, `replicateP n py` expects, as a printer, a list `l` of length `n`: if `n == 0`, there is nothing to print; if `n > 0`, `comap head` extracts the head of `l` to print it with `py`, and `comap tail` extracts its tail, of length `n-1`, to print it recursively.

```
replicateP :: Int → P u v → P [u] [v]
replicateP 0 py = return []
replicateP n py = do
  y  ← comap head py
  ys ← comap tail (replicateP (n - 1) py)
  return (y : ys)
```

We can now fulfill our task:

```
string :: P String String
```

```
string = comap length int >>= λ n → replicateP n char
```

Interestingly, if we erase applications of `comap`, *i.e.*, we substitute every expression of the form `comap f py` with `py` and ignore the types, we obtain what is essentially the definition of a parser. This is because `comap f` is the identity on the parser component of `P`. Thus the biparser code closely resembles standard, idiomatic monadic parser code.

Despite its simplicity, the syntax of length-prefixed strings is notably context-sensitive. Thus the example makes crucial use of the monadic interface of biparsers: a value (the length) must first be extracted to dynamically delimit the rest of the string to be parsed. This contrasts with parser generators, *e.g.*, Yacc, and applicative parsers, which are mostly restricted to context-free languages.

Additional features Lookaheads and backtracking are two notable features that make programming parsers much more convenient, although they complicate greatly the formulation of round-tripping properties. Thus, this presentation uses a very simplistic type of parsers for the sake of clarity; the library we mentioned in our contributions supports those features.

2.2 Monadic profunctors

The examples of the previous section make crucial use of monadic operations and `comap`. We call types implementing these combined operations *monadic profunctors*, and describe them in detail here.

Biparsers were defined via a data type with two type parameters, say `p u v`, which is functorial and monadic in the second parameter and *contravariantly* functorial in the first parameter. Contravariant functoriality in the first parameter was witnessed by the `comap` operation in the previous section, which we group under the following class `Cofunctor`:

```
class Cofunctor p where  comap :: (u → u') → p u' v → p u v
```

which should obey laws: `comap id = id` and `comap (f ∘ g) = comap g ∘ comap f`.

Definition 1. A two-parameter type `p` which is functorial in both its type parameters is called a *bifunctor*. In category theory, a bifunctor $\mathsf{F} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ is called a *profunctor*. In Haskell, the term *profunctor* has come to mean any bifunctor which is contravariant in the first type parameter and covariant in the second,[4] not just those that map to the category of sets and functions. This is the meaning of *profunctors* we adopt in this work.

Definition 2. A *monadic profunctor*, or *promonad*, is a profunctor `p` such that `p u` is a monad for all `u`. In terms of type class constraints, this means there is an instance `Cofunctor p` and for all `u`, there is a `Monad (p u)` instance. That universally quantified constraint(Bottu et al., 2017) can currently be expressed via the

[4] `http://hackage.haskell.org/package/profunctors/docs/Data-Profunctor.html`

constraint constructor `ForallF` (from the constraints package[5]) as `ForallF Monad p`. Thus, we capture promonads via the following class (which inherits all its methods from its superclasses):

```
class (Cofunctor p, ForallF Monad p) ⇒ Promonad p where
```

Monadic profunctors must obey the following laws about the interaction between cofunctor and monad operations:

```
comap f (return y)     =   return y
comap f (py >>= kz)    =   comap f py >>= (λ y → comap f (kz y))
```

These laws are equivalent to saying that `comap` lifts pure functions into monad morphisms. In Haskell, these laws are obtained *for free* by parametricity (Wadler, 1989), this means that every contravariant functor and monad is in fact a lawful monadic profunctor:

```
instance (Cofunctor p, ForallF Monad p) ⇒ Promonad p where
```

Corollary 1. Biparsers form a promonad as there is an instance of `Monad (P u)` and `Cofunctor p` satisfying the requisite laws.

2.3 Constructing monadic profunctors

Our examples share monadic profunctors as an abstraction, making it possible to write different kinds of bidirectional transformations monadically. Underlying these definitions of promonads is a common structure, which we explain here on biparsers, and which will be replayed in the future sections (Section 4 for lenses and Section 5 for *bigenerators*).

There are two simple ways in which a covariant functor `m` (resp. a monad) gives rise to a profunctor (resp. monadic profunctor). The first is by constructing a profunctor in which the first contravariant parameter is discarded, *i.e.*, `p u v = m v`; the second is as the function type from the contravariant parameter `u` to `m v`, *i.e.*, `p u v = u → m v`. We call these two constructions `Astr` and `Star` respectively:

```
newtype Astr m u v = Astr { unAstr :: m v }
newtype Star m u v = Star { unStar :: u → m v }
```

The `Star` type appears in the Haskell profunctors package; `Astr` is named as a pun to resemble `Star`, as the two are meant to respect domain-specific inverse relations in our framework.

[5] `http://hackage.haskell.org/package/constraints`

```
instance Monad m                          instance Monad m
      ⇒ Monad (Astr m u) where                  ⇒ Monad (Star m u) where
  return y = Astr (return y)                return y = Star (λ _ → return y)
  Astr py >>= kz = Astr                     Star py >>= kz = Star (λ x →
    (py >>= unAstr ∘ kz)                      py x >>= λ y → unStar (kz y) x)

instance Cofunctor (Astr m) where         instance Cofunctor (Star m) where
  comap f (Astr py) = Astr py               comap f (Star py) = Star (py ∘ f)
```

The product of two monadic profunctors is also a monadic profunctor. This follows from the fact that the product of two monads is a monad and the product of two contravariant functors is a contravariant functor.

```
data (:*:) p q u v = (:*:) { pfst :: p u v,  psnd :: q u v }

instance (Monad (p u), Monad (q u)) ⇒ Monad ((p :*: q) u) where
  return y = return y :*: return y
  py :*: qy >>= kz = (py >>= pfst ∘ kz) :*: (qy >>= psnd ∘ kz)

instance (Cofunctor p, Cofunctor q) ⇒ Cofunctor (p :*: q) where
  comap f (py :*: qy) = comap f py :*: comap f qy
```

We can redefine biparsers in terms of the above data types, its instances, and two standard monads: state and writer monads:

```
type State  s a = s → (a, s)
type Writer w a = (w, a)
type Biparser   = Astr (State String) :*: Star (Writer String)
```

The promonad definition for biparsers then comes for free from the constructions in this section. This construction could be reused to define a promonad for lenses in Section 4 and bigenerators in Section 5. However, we shall keep the presentation simple with the specialized instance definitions in each case. Lenses will actually require a small change in order to prevent conflicts due to the `Writer` component possibly overwriting previous actions.

Another simple promonad will be useful throughout the next sections.

Corollary 2. The type of pure functions, `(→)` is a promonad.

Proof. `(→)` is isomorphic to the `Star` of the identity monad.

2.4 Codec

The codec library (Chilton) provides a general type for bidirectional programming called `Codec`, isomorphic to our composite type `Astr r :*: Star w`.

```
data Codec r w c a = Codec { codecIn :: r a, codecOut :: c → w a }
```

Though the original codec library was developed independently, its current form is a result of this work. Particularly, we contributed to the package by generalising its original type (with `codecOut :: c → w ()`) to the one above, and provided `Monad` and `Profunctor` instances to support monadic bidirectional programming.

3 Compositionality and reasoning

The monadic profunctor structure provides a compact, natural way of defining biparsers using one set of syntax and operations. This approach also makes it easier to reason about properties relating the parsers and printers.

Recall that in Section 1.1, we discussed possible round-tripping properties of bidirectional parser/printer pairs. To talk more formally about these round-tripping properties of biparsers, it will be useful to separate the string and value components of the printer.

```
printS :: P u v → u → String          printV :: P u v → u → v
printS p x = fst (print p x)          printV p x = snd (print p x)
```

Then we can define the following properties:

Definition 3. A biparser `p :: P u u` is *left round-tripping* if for every input `x`, we have `parse p (printS p x) = (x, "")`.

Printing a value then parsing the output yields the same value. The printed string is also entirely consumed, *i.e.*, the printer does not produce superfluous content. We may naturally define a symmetrical property.

Definition 4. A biparser `p :: P u u` is *right round-tripping* if for every input string `s`, we have `parse p s = (x, "")` for some `x`, and `print p x = s`.

The presence of left or right round-tripping properties boils down to whether we put a "canonicity" requirement on the text or the AST representation of programs, restricting either the parser or the printer. In our example, and in many applications, the right round-trip property is too strong: it requires that every string is parseable and is the only representation of the resulting parsed value `x`. Therefore, our biparsers are designed only to respect left round-tripping, though the reasoning techniques we are going to present are not specific to this choice.

We may try to prove such a property directly by inspecting and expanding the definition of `string`, its auxiliary functions, and the underlying combinators. But this approach is not scalable: the proof becomes unwieldly for more complex biparsers. A more structured alternative is to generalize the properties to be *compositional*, *i.e.*, compatible with the promonad combinators `return`, `(>>=)`, and `comap`: if the primitives (here, `char`) satisfy such a property, then composite programs also satisfy the property.

For a promonad `p`, we consider a property $\mathcal{R}$ of promonadic values as an indexed subset of values $\mathcal{R}^{\mathtt{u}}_{\mathtt{v}} \subseteq \mathtt{p\ u\ v}$. Note that our definitions are in the context of Haskell where functions are not necessarily total. When `k y` is not defined, we consider `(k y)` $\in \mathcal{R}^{\mathtt{u}}_{\mathtt{w}}$ to be true. In other words, any promonadically compositional property is implicitly satisfied by an always failing action, represented by an undefined value $\bot$.

Definition 5. A property $\mathcal{R}$ is *promonadically compositional* with respect to a promonad `p` if the monad and profunctor operations are closed under $\mathcal{R}$, *i.e.*, the following conditions hold for all types `u`, `v`, `w`:

1. For all `y :: v`, $\texttt{(return y)} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}$ (comp-return)
2. For all `py :: p u v` and `k :: v → p u w`,
$$\left(\texttt{py} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}\right) \wedge \left(\forall \texttt{y}.\, \texttt{(k y)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}}\right) \implies \texttt{(py >>= } \lambda\ \texttt{y} \to \texttt{k y)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}}$$
(comp-bind)
3. For all `py :: p u' v` and `f :: u → u'`,
$$\texttt{py} \in \mathcal{R}^{\texttt{u'}}_{\texttt{v}} \implies \texttt{(comap f py)} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}$$ (comp-comap)

In other words, a promonadically compositional property is a congruence with respect to the promonad operations. Consequently, we can restrict the promonad operations to values satisfying property $\mathcal{R}$, yielding a promonad on $\mathcal{R}$. In fact, the inclusion $\mathcal{R}^{\texttt{u}}_{\texttt{v}} \subseteq \texttt{p u v}$ can be seen as a promonad morphism from $\mathcal{R}$ to `p`.

3.1 A compositional property of biparsers

Definition 6. A biparser `p :: P u v` is *well-behaved* if for all `x :: u` and `s :: String`, if `print p x` is defined, then:

```
parse p (printS p x ++ s) = (printV p x, s)
```

In other words, the printed string can be parsed back to the value returned by the printer. The universal quantification on strings `s` generalizes the property sufficiently for sequential composition `(>>=)` to preserve it.

Proposition 1. Well-behavedness is promonadically compositional with respect to the biparser implementation of Section 2.

Proposition 2. The primitive biparser `char` is well-behaved. As a corollary, so is `string`.

Programming against an interface of well-behaved primitives and combinators which preserve well-behavedness ensures that the resulting programs are well-behaved by construction.

3.2 Alignment

We have managed to guarantee the well-behavedness of biparsers through the monadic interface. But by itself, well-behavedness is not sufficient for round-tripping because of a loophole: if the printer always fails, then the biparser is vacuously well-behaved. The following property, which we call *alignment*, rules out such behaviour (so that a biparser at least sometimes succeeds) and relates the output of a printer to its input:

Definition 7. Biparsers of type `p u u` for a type `u`, are *aligned* if `x = printV p x`, *i.e.*, `printV p` is the identity function.

Corollary 3. An aligned and well-behaved biparser is left round-tripping.

Proof. By alignment, `(printV p x, s) = (x, s)`. By well-behavedness, `parse p (printS p x ++ s) = (printV p x, s)`. Conclude with `s = ""`.

However, alignment is not a compositional property as only it applies to biparsers of type `P u u`, with the same input and output types, but the types of `return`, `(>>=)` and `comap` allow them to vary independently. Furthermore, the totality it implies suggests that alignment should not be thought of as an invariant of biparsers (compositionality being one way to formalize such invariants): generally, composing aligned biparsers does not necessarily produce aligned biparsers, and composing non-aligned biparsers does not necessarily produce non-aligned ones.

Instead, we can use equational reasoning to establish alignment. Recall that for a biparser `P u v`, `printV` extracts a partial function `u → v`:

```
printV :: P u v → (u → v)
printV p x = snd (printer p x)
```

`printV` is in fact a promonad morphism, *i.e.*, it satisfies the following equations, where promonadic operations on the left belong to the promonad `P`, and those on the right belong to `(→)`:

```
printV (return y) = return y
printV (py >>= λy → kz y) = printV py >>= λy → printV (kz y)
printV (comap f py) = comap f (printV py)
```

These equations enable a proof that the `string` biparser is aligned.

Proposition 3. The `string` biparser is *aligned*, *i.e.*, `printV string` is the identity function on strings.

Proof. Starting with the definition of `string`, applying the above equations from left to right, then expanding the definitions of promonadic operations on `(→)`, we obtain the following:

```
printV string
  = do n ← comap length (printV int)
       printV (replicateP n char)
  = λx → printV (replicateP ((printV int) (length x)) char) x
```

We may prove the following lemmas:

```
printV int = id
printV (replicateP (length u) p) u = map (printV p) u
```

Then the above reduces to

```
printV string = (λ x → x) = id
```

Corollary 4. The biparser `string` satisfies the left round-tripping property.

Thus, we have reduced the proof of the left round-tripping property to a proof of alignment and a proof of well-behavedness on the base biparsers (well-behavedness is then guaranteed by compositionality).

3.3 Quasicompositionality

In addition to equational reasoning for alignment, there is another regularity of biparser construction which we can explore. In particular, all occurrences of `(>>=)`, which also appear in the form of syntactic sugar `"←"` as part of Haskell's **do**-notation, are accompanied by a preceding `comap`.

Considering a biparser `(comap f py >>= λ y → kz y)`, we can imagine `py` informally as a biparser for a "header" `y`, which contains information about how to subsequently parse or print the "body" using `kz y`. The final value contains both the header and the body; when interpreting the biparser as a printer, we use the function `f` to extract the header from it, print it with `py`, and pass it along to `kz`. Thus, `kz y` is a biparser specifically for values which have `y` as their header. We call `kz` an *injective (Kleisli) arrow*.

Definition 8. Let `m` be a monad. A function `k` of type `v → m w` is an *injective arrow* if there exists a function `f` of type `w → v`, which we call *a sagittal left inverse* of `k`, such that, for all `y`:

```
(k y >>= λz → return (y, z))   =   (k y >>= λz → return (f z, z))
```

Informally, an injective arrow represents a computation producing an output `z` from which the "header" `y` that is an input to the arrow can be extracted back. As the naming suggests, injective arrows generalize injective functions. Indeed, a function `f` is injective iff there exists `f'` (a *left inverse*) such that `f' ∘ f = id`. This is equivalent to the above condition in the identity monad (`m w ~ w`).

Note that all but one of the right operands of `(>>=)` that can be found in our examples are injective arrows, *e.g.*, `(λ n → replicateP n char)`.

The one arrow that is not injective is `λ ds → return (read ds)` in the `int` function. Indeed, the `read` function is not injective, since multiple strings may parse to the same integer: `read "0" = read "00" = 0`. It may only be considered injective after taking a quotient of the set of input strings by the equivalence between strings which differ only in a prefix of zeroes.

The pattern where every `(>>=)` is accompanied by a `comap` points towards the following notion of *quasicompositionality*.

Definition 9. Let `p` be a monadic profunctor. A property $\mathcal{Q}_{\mathsf{u}} \subseteq \mathtt{p\ u\ u}$ is *quasicompositional* if the following holds, for all types `u`, `v`:

1. For all `x :: u`, $\mathtt{(return\ x)} \in \mathcal{Q}^{\mathsf{u}}$.
2. For all `px :: p u u`, `ky :: u → p v v`. and sagittal left inverse `f :: v → u` of `ky`,

$$\mathtt{px} \in \mathcal{Q}^{\mathsf{u}} \wedge (\forall \mathtt{x}\ .\ \mathtt{(ky\ x)} \in \mathcal{Q}^{\mathsf{v}}) \implies \mathtt{(comap\ f\ px >{=}\ \lambda\ x \to ky\ x)} \in \mathcal{Q}^{\mathsf{v}}.$$

One example of quasicompositional property is a partial variant of the left round-tripping property which is conditional on the success of the printer. We call the previous round-tripping properties *total* to distinguish them from this one.

Definition 10. A biparser `p` is *partial left round-tripping* when the following holds: if `printerS p x = s`, then `parser p (s ++ s') = (x, s')` for all `s'`.

Quasicompositionality is another way to structure reasoning about round-tripping properties, other than combining well-behavedness and alignment. Although the partial left round-tripping property is weaker, it may be possible to design an abstraction to compose injective arrows, akin to the *partial isomorphisms* of Rendel and Ostermann (2010) (also called partial injective functions) and thus to obtain quasicompositional properties by construction. However, our focus here is to reuse the existing interface for monads as much as possible.

4 Monadic bidirectional programming for lenses

Recall the standard characterisation of lenses as a pair of functions ($\texttt{get} : S \to V, \texttt{put} : S \to V \to S$) potentially satisfying the laws of *well-behaved lenses* shown in the introduction. Following a similar scheme to the "monadisation" of parsers and printers, we define the following new data type for lenses:

```
data L s u v = L { get :: s → v, put :: s → u → (s, v) }
```

The type of `get` is the same as before, but `put` has changed slightly. Instead of mapping a source and a view to a source, `put` now maps a source `s` and a different type `u`, which we call a *pre-view*, into a source `s` paired with a view `v`.

We split the source and view components of `put` (which are partial functions):

```
putS :: L s u v → (s → u → s)        putV :: L s u v → (s → u → v)
putS l s x = fst (put l s x)         putV l s x = snd (put l s x)
```

Similarly to biparsers, a pre-view `x :: u` can be understood as *containing* the view `y :: v` that is to be merged with the source, and returned together with the updated source. Ultimately, we wish to form lenses of matching input and output types `u ~ v`, satisfying the standard lens laws.

```
get l (putS l s x) = x                    (L-PutGet)
putS l s (get l s) = s                    (L-GetPut)
```

For every source type `s` and pre-view type `u`, the lens type `L s u` is a monad:

```
instance Monad (L s u) where
  return :: v → L s u v
  return y = L (λ _ → y) (λ s _ → (s, y))

  (>>=) :: L s u v → (v → L s u w) → L s u w
  ly >>= kz = L getter putter where
    getter s = get (kz (get ly s)) s
    putter s x = let (s', y) = put ly s x
                 in put (kz y) s' x
```

The lens `return y` always gets and puts the view `y`, leaving the source untouched. We also have a way to map over pre-views:

```
comap :: (u → u') → L s u' v → L s u v
comap f py = L (get py) (λ s x → put py s (f x))
```

Moreover, lenses can be composed to access nested views. Given a lens `lt` to view `s` as `t`, and a lens `ly` to view `t` as `v`, the combinator `(>>>)` creates a lens to view `s` as `v`; lenses actually form a category; the signature here is more general and allows the pre-view and view types of the inner lens to differ:

```
(>>>) :: L s t t → L t u v → L s u v
lt >>> ly = L getter putter where
  getter = get ly ∘ get lt
  putter s x = let t          = get lt s
                   (t', y)    = put ly t x
                   (s', _t'') = put lt s t' in (s', y)
```

As an example of programming with these monadic lenses, we consider lenses over the following data type of binary trees labeled by integers.

```
data Tree = Leaf | Node Tree Int Tree
```

We define two primitive "shallow" lenses: one lens accesses the label at the root if it is a `Node`, otherwise returning `Nothing`; another lens accesses the right child. The second lens is partial and assumes the root constructor is a `Node`.

```
rootL :: L Tree (Maybe Int) (Maybe Int)
rootL = L getter putter where
  getter t = case t of
    Leaf       → Nothing
    Node _ n _ → Just n
  putter t n' = case (t, n') of
    (_, Nothing)   → (Leaf, n')
    (Leaf, Just n) → (Node Leaf n Leaf, n')
    (Node l _ r, Just n) → (Node l n r, n')
```

```
rightL :: L Tree Tree Tree
rightL = L getter putter where
  getter (Node _ _ r) = r
  putter (Node l n _) r =
    (Node l n r, r)
```

Composing these primitives, we obtain a lens to view the right spine of a tree. As a `get`, it first views the root of the source tree through `rootL` as `hd`, and whether it recurse or not depends on whether it is a node (with label `n`) or a leaf, using `rightL` to shift the context. As a `put`, it updates the root using the head of the list, which is returned as the view `hd`, and continues with the same logic.

```
spineL :: L Tree [Int] [Int]
spineL = do
  hd ← comap headM rootL
  case hd of
    Nothing → return []
    Just n → do
      tl ← comap tail (rightL >>> spineL)
      return (n : tl)
```

This auxiliary function safely gets the head of a list, if it exists.

```
headM :: [a] → Maybe a
headM (a : _) = Just a
headM [] = Nothing
```

To illustrate the action of this lens, consider a tree:

```
t0 = Node (Node Leaf 0 Leaf) 1 (Node Leaf 2 Leaf)
```

Getting the left spine (`get spineL t0`) yields the list `[1, 2]`. The tree spine can be updated to `[3, 4, 5]` yielding the tree:

```
fst (put spineL t0 [3, 4, 5])
  = Node (Node Leaf 0 Leaf) 3 (Node Leaf 4 (Node Leaf 5 Leaf))
```

4.1 Compositionality and reasoning for lenses

Section 3 introduced notions of *compositional* and *quasicompositional* properties for promonads, showing properties of well-behavedness (compositional), alignment, and partial round-tripping (quasicompositional) for biparsers. We follow the same scheme here for lenses. Firstly, lens combinators (promonadic operations and `(>>>)`) have a compositional property of *well-behavedness* which parallels a similar notion for biparsers. Following this, we consider lens alignment.

Well-behaved lenses The standard properties of a well-behaved lens capture the round-tripping relationship between the put and get operations (L-PutGet and L-PutGet, p. 15). We adapt these properties to our promonadic form of lenses. This adaption takes account of the change to `put`, which now outputs a view rather than taking it as an input; the pre-view argument to `put` takes the place of the view, but the output view should related to the pre-view.

Definition 11. A lens `l :: L s u v` is *well-behaved* if the following holds:

$$\texttt{put l s x = (s', y)} \implies \texttt{get l s' = y} \quad \text{(L-semi-PutGet)}$$

$$\texttt{get l s = y} \wedge \texttt{put l s x = (s', y)} \implies \texttt{s' = s} \quad \text{(L-semi-GetPut)}$$

If putting `y` in `s` (from the pre-view `x`) succeeds and results in `s'`, then we get `y` from `s'` (L-semi-PutGet); if we get `y` from `s`, then putting `y` in `s` results in `s` unchanged (L-semi-GetPut). However, to put a given view `y`, we must first have a pre-view `x` which contains it, this is why, formally, `put` occurs in the premise of the latter implication. Moreover, this definition of well-behavedness is weaker than that which can be found in the lens literature, due to the decoupling between views and pre-views, hence the "semi-" prefix.

One condition of compositionality that merits particular attention is (comp-bind): if the lens `ly :: L s u v` is well-behaved, and if `kz y :: L s u w` is well-behaved for all `y`, then `ly >>= kz` is a well-behaved lens. However, the definition of `(>>=)` above is not compatible with (L-semi-PutGet). Indeed, to compose the two lenses, their `put` functions can only be applied one after the other, on different sources `s` and `s'`. The second modification may break the relation between the source `s'` and its view `y` through the first lens `ly`.

For instance, consider a simple lens on the first component of a pair.

```
fstL :: L (u, v) u u
fstL = L getter putter where
  getter (x, _) = x
  putter (_, y) x = ((x, y), x)
```

Composing it sequentially with itself yields a lens that gets from and puts into the first component twice.

```
fstTwiceL :: L (u, v) (u, u) (u, u)
fstTwiceL = fstL >>= λ x → fstL >>= λ x' → return (x, x')
```

If we put two distinct elements with that lens, the second one overwrites the first, so that we get it back twice. Thus, it fails the (L-semi-PutGet) law.

```
put fstTwiceL (x, y) (x1, x2) = ((x2, y), (x1, x2))
get fstTwiceL (x2, y) = (x2, x2)
```

This is actually not surprising; it is well known in the bidirectional transformation literature that if a source variable is duplicated in the view, all the occurrences of it must remain equal amid updates (otherwise PutGet will be violated) (Foster et al., 2007).

Instead, we can enforce the (L-semi-PutGet) law by trading totality for well-behavedness: we add a run-time assertion that the view through the first lens remains untouched by the second lens. This requires the type of views to have decidable equality, represented by the `Eq` type class:

```
(>>=) :: Eq v ⇒ L s u v → (v → L s u w) → L s u w
ly >>= kz = L getter putter where
  getter s = get (kz (get ly s)) s
  putter s x = let (s', y) = put ly s x
                   (s'', z) = put (kz y) s' x
               in if get ly s'' == y then (s'', z)
                                     else error "Put␣conflict"
```

The condition `get py s'' == y` means that, after the first lens `py` puts `y` into the source, whatever `kz y` puts must not modify `y` that is already there. When this condition does not hold, we say that `py` and `kz` are *in conflict*.

Note, that despite the equality contraint, this definition can still be made into an instance of `Monad`, for example using the approach of Sculthorpe et al. (2013), but this is an orthogonal issue to the core idea.

Proposition 4. Well-behavedness is promonadically compositional with respect to the promonad defined by `return` and the new `(>>=)`.

However, to account for the `(>>>)` operator, we need a notion of alignment.

Alignment As before, verifying the round-tripping between `get l` and `put l` is reduced to checking that an interpretation of the lens `l`, as a pure function `putV l s` from pre-views to views, is the identity (Section 3.2).

Definition 12. A lens `l :: L s u u` is *aligned* if `putV l s` be the identity function for all sources `s`.

Proposition 5. Well-behavedness and alignment imply the lens laws L-GetPut and L-PutGet.

However, we find that the mapping from `l` to `putV l s` does not define a promonad morphism, which prevents us from applying the equational reasoning

techniques we demonstrated for parsers. The issue is caused by the conditional expression we inserted in the `put` component of `(>>=)`.

Consider again the (ill-behaved) `fstTwiceL` lens we introduced earlier. Note that `putV fstL s = id` — nothing goes wrong if we just update the first component of a pair. Thus, if the function `λ l → putV l s` defined a promonad morphism for all `s`, we would have:

```
putV fstTwiceL s = λ (x, x') → (putV fstL s x, putV fstL (putS fstL s) x')
                 = λ (x, x') → (x, x')
```

Yet, by our design, `fstTwiceL` should fail whenever `x /= x'`, so we cannot have a promonad morphism. We have the following equation for `ly >>= kz` *only if* `ly` and `kz` are not in conflict.

```
putV (ly >>= λ y → kz y) s = putV ly s >>= λ y → putV (kz y) s
```

Theoretically, avoiding conflicts requires a non-trivial level of non-local reasoning that is outside the scope of this article. In practice, a common pattern is to sequentially compose accesses to distinct fields of a constructor, for example:

```
data Record = Record { x :: X, y :: Y }

xLens :: L Record X X
yLens :: L Record Y Y

twoDisjointAccesses :: (v → w → t) → L X u v → L Y u w → L Record u t
twoDisjointAccesses f ly lx = do
  x ← xLens >>> ly
  y ← yLens >>> lz
  return (f x y)
```

It is clear that the two lenses `xLens >>> ly` and `yLens >>> lz` are not in conflict.

Nevertheless, we can still guarantee *conditional alignment*: if `put ly` succeeds without conflicts on some input `x`, then `putV ly s` maps `x` to itself, and the Put-Get and GetPut laws hold for it. More formally, consider the domain-theoretic denotations of programs, where the smallest element `_|_` corresponds to undefined or error values. In the definition of `(>>=)`, the conditional expression `(if ... then (s'', w) else error "...")` is denotationally smaller than `(s'', w)`. With that ordering defined between partial values, written `(-<)`, `putV` defines a *submorphism of monads*:

```
(putV (ly >>= kz) s)   -<   (putV ly >>= λy → putV (kz y) s)
```

This approximation abstracts away the concern of put-conflicts to recover a form of (in)equational reasoning.

Definition 13. A lens `l` is *partially aligned* if, for all `s`, we have `putV l s -< id`. In other words, if `putV l s x = x'` for some `x` and `x'`, then `x = x'`.

Proposition 6. If `lt :: L s t t` is partially aligned and well-behaved, and if `ly :: L t u v` is well-behaved, then `lt >>> ly` is well-behaved.

Proposition 7. The primitive lenses `rootL` and `rightL` are well-behaved. As a consequence, the lens `spineL` we wrote is well-behaved.

Partial alignment implies the "lens laws up to conflicts", L-PutGet-Ineq and L-GetPut-Ineq. Indeed, `spineL` satisfies them. We also note that these two laws are quasicompositional:

```
get l (putS l s y) -< y          (L-PutGet-Ineq)
putS l (get l s) -< s            (L-GetPut-Ineq)
```

5 Monadic bidirectional programming for generators

We capture the notion of *bigenerators* (*bidirectional generators*) as the type:

```
data G u v = G { generate :: Gen v, check :: u → Maybe v }
```

This extends the notion of generators in property-based testing frameworks like QuickCheck (Claessen and Hughes, 2000) to a bidirectional setting. A `G u v` value represents a *set of generable values*, where `generate` is a generator of views `v` in that set and `check` maps pre-views `u` to members of the generated set, inducing a predicate on pre-views through explicit partiality, modelled by `Maybe`.

A generator of values `v` and a predicate on `v` (modelled by `v → Bool`) together define a bidirectional generator with the same pre-view and view type, via `mkG`. A bigenerator can be mapped to a generator via `generate` above and a predicate via `toPredicate` below.

```
mkG :: Gen v → (v → Bool) → G v v
mkG generate predicate = G generate check where
  check y = if predicate y then Just y else Nothing

toPredicate :: G u v → u → Bool
toPredicate g x = isJust (check g x) where
  isJust (Just _) = True
  isJust Nothing  = False
```

Our bigenerator type has the structure of a promonad, with the definition:

```
instance Monad (G u) where
  return y = G (return v) (λ _ → Just y)
  py >>= kz = G generate check where
    generate = generate py >>= λy → generate (kz y),
    check x = check py x >>= λy → check (kz y) x

instance Cofunctor G where
  comap :: (u → u') → G u' v → G u v
  comap f py = G (generate py) (check py ∘ f)
```

We define two primitives: `bool` generates a boolean according to a Bernoulli distribution with a given parameter $p \in [0, 1]$, `inRange` generates an integer according to the uniform distribution in a given range. As predicates, `bool` makes

no assertion, `inRange` checks that the input integer is within the given range.

```
bool :: Double → G Bool Bool
bool p = mkG
  (fmap (< p) (choose (0, 1)))
  (λ_ → True)
```

```
inRange :: (Int, Int) → G Int Int
inRange (min, max) = mkG
  (choose (min, max))
  (λx → min ≤ x && x ≤ max)
```

We consider again a type of labelled trees, with some field accessors. On the bottom right, `leaf` is a simple bigenerator for leaves.

```
data Tree = Leaf | Node Tree Int Tree

nodeValue :: Tree → Int
nodeValue (Node _ n _) = n

nodeLeft, nodeRight :: Tree → Tree
nodeLeft (Node l _ _) = l
nodeRight (Node _ _ r) = r
```

```
isLeaf :: Tree → Bool
isLeaf Leaf = True
isLeaf (Node _ _ _) = False

leaf :: G Tree Tree
leaf = mkG (return Leaf) isLeaf
```

We then define a specification of binary search trees (`bst` below), *i.e.*, trees whose nodes are in sorted order. A corresponding generator and predicate are extracted on the right from this bigenerator:

```
bst :: (Int, Int) → G Tree Tree
bst (min, max) | min > max = leaf
bst (min, max) = do
  isLeaf' ← comap isLeaf (bool 0.5)
  case isLeaf' of
    True → return Leaf
    False → do
      n ← comap nodeValue (inRange (min, max))
      l ← comap nodeLeft  (bst (min, n - 1))
      r ← comap nodeRight (bst (n + 1, max))
      return (Node l n r)
```

```
genBST :: Gen Tree
genBST =
  generate (bst (0, 20))

checkBST :: Tree → Bool
checkBST =
  toPredicate (bst (0, 20))
```

The bigenerator `bst` is parameterized by an integer interval restricting the possible values of the trees. As a random generator, we flip a coin (`bool 0.5`) to decide whether to generate a leaf or a node. In the case of a node (the `False` branch), we first generate a value for the root, which is used to update the bounds of the left and right recursive cases.

As a predicate, `bst` first checks whether the root is a leaf (`isLeaf`); returning a boolean allows us to reuse the same case expression as for the generator. If it is a node, we check that the value is within the given range and then recursively check the subtrees.

5.1 Compositionality and reasoning for bigenerators

A random generator can be interpreted possibilistically as the set of values it may generate, while a predicate represents the set of values satisfying it. For a bigenerator `py`, we write `x ∈ generate py` when `x` is a possible output of the generator `generate py`.

Generators should match the predicate they represent. We state this requirement as two round-tripping properties: a bigenerator is *sound* if every value which it can generate satisfies the predicate; a bigenerator is *complete* if every value which satisfies the predicate can be generated. As discussed in the introduction, completeness is more important than soundness in testing, because unsound tests will be filtered out by the predicates as usual, but completeness determines the potential adequacy of testing.

It turns out that completeness is much easier to enforce constructively for bigenerators, whereas soundness is harder to ensure in our framework.

Completeness Completeness is defined as follows:

Definition 14. A bigenerator `g :: G u u` is *complete* when `toPredicate g x = True` implies `x ∈ generate g`.

Similarly to biparsers and lenses, this round-tripping property can be split into compositional and pure fragments: *well-behavedness* and (partial) *alignment*.

Definition 15. A bigenerator `g :: G u v` is *well-behaved* when `check g x = Just y` implies `y ∈ generate g`.

Since the point of a predicate is to filter out invalid values, the notion of alignment we shall use for bigenerators should allow partial functions. The same lightweight reasoning methods as before still apply to verify completeness.

Definition 16. A bigenerator `g :: G u u` is *partially aligned* when `check g x = Just x'` implies `x = x'`.

Proposition 8. An aligned and well-behaved generator is complete.

Soundness Soundness is defined as follows:

Definition 17. A bigenerator `g :: G u u` is *sound* if for all `x :: u`, `x ∈ generate g` implies that `toPredicate g x = True`.

Soundness is not as easy to ensure as completeness in our framework. For instance, the following bigenerator is unsound:

```
unsound = comap (λ _ → 3) (inRange (0,2))
```

As a generator, this just produces an integer between 0 and 2 (`comap` has no effect). But as a predicate, the input value is ignored, and 3 is checked to be in the interval $[0, 2]$, which fails of course. Thus it is unsound: values are generated, but none of them satisfies the predicate. Furthermore, since `check unsound x = Nothing` for all `x`, this bigenerator is vacuously well-behaved and aligned.

As this example shows, we may lose information about the predicate by applying arbitrary functions with `comap`, whereas the generator remains untouched. We recover a close correspondence between generators and predicates with the restriction to composition patterns identified by quasicompositionality.

Proposition 9. Completeness and soundness are quasicompositional.

Proposition 10. `bool`, `inRange (n, m)` and `leaf` are sound and complete for all `(n, m)`. As a consequence, `bst (n, m)` is also sound and complete, for all `(n, m)`.

6 Discussion and Related Work

6.1 Applicative bidirectional programming

Existing techniques in the literature and in the wild have leveraged various other common abstractions to compose bidirectional programs, such as categories (Alimarine et al., 2006), and, more closely related to our work, applicative functors. While monadic sequential composition `(>>=)` allows one computation to depend on the result of a previous one, applicative composition can only sequence independent computations, whose results can nevertheless be combined purely. It is often complemented by a combinator for non-deterministic choice or backtracking in order to allow a limited but often effective form of data dependency.

Rendel and Ostermann (2010) proposed a bidirectional programming interface imitating applicative functors. Although its presentation was centered around parsers and printers, it is also applicable to the settings of lenses and random generators we explored here. To account for the opposite polarities of parsers and printers, which are covariant and contravariant functors, respectively, the interface expects user-defined "partial isomorphisms" to transform and combine parser outputs and deconstruct printer inputs in the same bidirectional program, as opposed to regular functions in "unidirectional" applicative programming.

Our framework adapts gracefully to applicative programming, which is in fact a restricted form of monadic programming. By separating the input type from the output type, we can reuse the existing interface of applicative functors without modifying it. The pattern of combining profunctors with applicative functors is actually folklore in the Haskell community, sometimes called *monoidal profunctors*. Besides its generalization to monads, the concepts of alignment and quasicompositionality for verifying round-tripping properties are novel to the best of our knowledge.

The codec (Chilton) library we mentioned in Section 2.2 prominently features two applications of this monadic programming style: binary serialization (a form of parsing/printing) and conversion to and from JSON structures (analogous to lenses above).

Opaleye (Purely Agile), an EDSL of SQL queries for Postgres databases, uses an interface of monoidal profunctors to implement generic operations such as transformations between Haskell datatypes and database queries and responses.

6.2 Lenses

Bidirectional transformations represent a widely applicable solution that is used and studied in many domains (Czarnecki et al., 2009). Among language-based

solutions, the lens framework is most influential (Foster et al., 2007; Barbosa et al., 2010; Bohannon et al., 2008; Foster et al., 2010; Rajkumar et al., 2013; Pacheco et al., 2014a).

Many different representations of lenses have been studied to make them easier to work with. *Profunctor optics* (Pickering et al., 2017) are a generalization of lenses to which our work bears a striking resemblance.

More precisely, a profunctor optic between a source type `s` and a view type `v` is a function of type `p v v → p s s`, for an abstract profunctor `p`. There are two interesting comparisons to make between profunctor optics and monadic profunctors. First, they offer orthogonal composition patterns: profunctor optics can be composed "vertically" simply using function composition, whereas promonadic composition is "horizontal". In both cases, composition in the other direction can only be obtained by breaking the abstraction, as we do with a custom `(>>>)` combinator. Second, promonadic definitions are loosely similar to profunctor optics. By abstracting over primitives, *e.g.*, `rootL` and `rightL` in Section 4, the type of a monadic lens such as `spineL` becomes:

```
L Tree (Maybe Int) (Maybe Int) → L Tree Tree Tree → L Tree [Int] [Int]
```

This corresponds loosely to a profunctor optic with source type `[Int]`, and two types of views: `Maybe Int` and `Tree`, corresponding to the second and third arguments to the type constructor `L`. This is not exclusive to promonadic lenses. Biparsers and bigenerators are also profunctor-optic-like in the same way, but we chose the example of lenses to better set our work apart from the point of view of profunctor optics. Indeed, the parallel we drew mismatches the notions of "view" and "source" in the two approaches. Among other differences, in the type of promonadic lenses `L s u v`, the source type `s` is internal to the concrete profunctor `L s`, whereas profunctor optics operate with an abstract profunctor `p` and rely on parametricity to preserve the relation between the view and source in `p v v → p s s`. Nevertheless, this resemblance suggests a deeper connection between monadic profunctors and profunctor optics to investigate in the future.

More dramatically, the framework of *applicative lenses* (Matsuda and Wang, 2015) uses a different function representation of lens to break out from the point-free restriction, and enable bidirectional programming with explicit recursion and pattern matching. Note that the use of "applicative" in applicative lenses refers to the transitional sense of programming with λ-abstraction and functional application, which is not related to *applicative functors* in Haskell (McBride and Paterson, 2008). In this paper, we construct a new composition framework for lenses based on monads, which fits neatly into Haskell's existing type class hierarchy. This new composition technique not only enables lens programming in the monadic style, but also opens up bidirectional programming to new domains that utilise monadic composition, *e.g.*, random generators for testing.

The work on *monadic lenses* (Abou-Saleh et al., 2016) investigates lenses with effects. For instance, a put could require additional input to resolve conflicts when merging a view and a source. Representing effects with monads helps to reformulate the laws that define well-behaved lenses. In contrast, we made the type of lenses itself a monad, and we showed how they can be composed monad-

ically to preserve well-behavedness. Our method is applicable to monadic lenses, yielding what one might call *monadic monadic lenses*: monadically composable lenses with monadic effects. We conjecture that laws for monadic lenses can be adapted to this setting with similar compositionality properties.

6.3 Random generators and predicates

Previous approaches to unifying random generators and predicates mostly focused on deriving generators from predicates. One general technique for instance consists in evaluating predicates lazily to drive generation (random or enumerative) (Boyapati et al., 2002; Claessen et al., 2015), but one loses control over the resulting distribution of generated values. *Luck* (Lampropoulos et al., 2017) is a domain-specific language blending narrowing and constraint solving to specify random generators as predicates with user-provided annotations to control the probability distribution.

In contrast, our programs can be viewed as generators annotated with left inverses with which to derive predicates. This reversed perspective comes with trade-offs: high-level properties would be more naturally expressed in a declarative language of predicates, whereas it is a priori more convenient to implement complex generation strategies in a specialized framework for random generators.

7 Conclusions

This paper advanced the expressive power of bidirectional programming; we showed that the classic bidirectional patterns of parsers/printers and lenses can be restructured in terms of *monadic profunctors* to provide sequential composition, with associated reasoning techniques. This effectively opens up a new area in the design of embedded domain-specific languages for bidirectional programming, that does not restrict programmers to stylised interfaces. Our example of generators broadened the scope of bidirectional programming from transformations (converting between two data representations) to non-transformational applications.

However, this is not the final word on sequentially composable bidirectional programs. In all three applications, round-tripping properties are similarly split into well-behavedness, which is weaker than the original property but compositional, and alignment, which is equationally friendly. An open question is whether an underlying structure can be formalized, perhaps based on an adjunction model, that captures more concretely bidirectional programs than monadic profunctors. Moreover, partiality is pervasive in our development, and the composition patterns captured by quasicompositionality require discipline and are still prone to error. We hope to develop tools and methods, such as type systems to automatically ensure correct usage of these patterns and guarantee compositional properties.

Bibliography

F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. *Reflections on Monadic Lenses*, pages 1–31. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_1.

A. Alimarine, S. Smetsers, A. Weelden, M. Eekelen, and R. Plasmeijer. There and back again: arrows for invertible programming. In *In Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97. ACM Press, 2006.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In P. Hudak and S. Weirich, editors, *ICFP*, pages 193–204. ACM, 2010. DOI: 10.1145/1863543.1863572.

A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347. ACM, 2006.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In G. C. Necula and P. Wadler, editors, *POPL*, pages 407–419. ACM, 2008. ISBN 978-1-59593-689-9.

G.-J. Bottu, G. Karachalias, T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler. Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 148–161, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5182-9. DOI: 10.1145/3122955.3122967.

C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133. ACM, 2002. ISBN 1-58113-562-9. DOI: 10.1145/566172.566191.

P. Chilton. `codec` library. https://hackage.haskell.org/package/codec.

K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. DOI: 10.1145/351240.351266.

K. Claessen, J. Duregård, and M. H. Palka. Generating constrained random data with uniform distribution. *Journal of functional programming*, 25, 2015. ISSN 0956-7968.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. *Bidirectional Transformations: A Cross-Discipline Perspective*, pages 260–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02408-5. DOI: 10.1007/978-3-642-02408-5_19.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *SSGIP*, volume 7470 of

Lecture Notes in Computer Science, pages 1–46. Springer, 2010. ISBN 978-3-642-32201-3.

G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437âĂŞ444, 1998.

L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 114–129, 2017.

K. Matsuda and M. Wang. Flippr: A prettier invertible printing system. In *European Symposium on Programming (ESOP)*, pages 101–120, March 2013.

K. Matsuda and M. Wang. Applicative bidirectional programming with lenses. In K. Fisher and J. H. Reppy, editors, *Proceedings of ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 62–74. ACM, 2015. ISBN 978-1-4503-3669-7. DOI: 10.1145/2784731.2784750.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 47–58. ACM, 2007. ISBN 978-1-59593-815-2.

C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In W.-N. Chin and J. Hage, editors, *PEPM*, pages 39–50. ACM, 2014a. ISBN 978-1-4503-2619-3. DOI: 10.1145/2543728.2543737.

H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *PPDP*. ACM, 2014b.

M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *The Art, Science, and Engineering of Programming*, 1(2), 2017. DOI: 10.22152/programming-journal.org/2017/1/7.

Purely Agile. opaleye library. https://hackage.haskell.org/package/opaleye.

R. Rajkumar, S. Lindley, N. Foster, and J. Cheney. Lenses for web data. In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty-printing. In J. Gibbons, editor, *Haskell*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0252-4.

N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *ACM SIGPLAN Notices*, volume 48, pages 287–298. ACM, 2013.

J. Voigtländer. Bidirectionalization for free! (pearl). In Z. Shao and B. C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009. ISBN 978-1-60558-379-2.

P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.