Composing bidirectional programs monadically

Li-yao Xia[1], Dominic Orchard[2], and Meng Wang[3]

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

Abstract. Software frequently converts data from one representation to another and vice versa. Naïvely specifying both conversion directions separately is error prone and introduces conceptual duplication. Instead, *bidirectional programming* techniques allow programs to be written which can be interpreted in both directions. However, these techniques often employ programming idioms that are alien to non-experts, via restricted, specialised combinator libraries. Instead, we introduce a framework for composing bidirectional programs monadically, enabling bidirectional programming with familiar abstractions in functional languages such as Haskell. We demonstrate the generality of our approach applied to parsers/printers, lenses, and generators/predicates. 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. A well-known example solves the *view-update problem* [**?**] from relational database design: a *view* is a derived database table, computed from concrete *source* tables by a query. The problem is to map an update of the view back to a corresponding update on the source tables. This is captured by a bidirectional transformation. The bidirectional pattern is found in a broad range of applications, including parsing [**? ?**], refactoring [**?**], code generation [**? ?**], and model transformation [**?**].

When programming a bidirectional transformation, one can separately construct the forwards and backwards functions. However, this approach duplicates effort, is prone to error, and causes subsequent maintenance issues. These problems can be avoided by using specialised programming languages that generate both directions from a single definition [**? ? ?**], a discipline known as *bidirectional programming*.

The most well-known language family for BX programming is *lenses* [**?**]. A lens captures transformations between *sources* S and *views* V via a pair of functions $\texttt{get} : S \to V$ and $\texttt{put} : V \to S \to S$. 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` makes 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* [**?** **?**] should satisfy:

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

Lens languages are conventionally designed to enforce 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. In this paper, we instead focus on expressing bidirectional programs directly, using monads as an interface for sequential composition. Monads are a popular pattern [**?**], sitting on top of the Haskell's functor/applicative/monad framework, which combinator libraries in other domains routinely exploit. By introducing monadic composition to BX programming, it not only significantly expands the expressiveness of BX languages, but also opens up a route for programmers to explore the connection between BX programming and mainstream uni-directional programming. Moreover, it appears that many applications of bidirectional transformations (e.g., parsers and printers [**?**]) do not share the lens *get*/*put* pattern, and as a result have not been sufficiently explored. However, monadic composition is known to be an effective way to construct at least one direction of such transformations (e.g., parsers).

Contributions In this paper, we deliberately avoid the well-tried approach of specialised lens languages, instead exploring 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, demonstrating how they can be subject to programming monadically. By being uncompromising about the monad interface, we expose the essential ideas behind our framework whilst maximising its usability. The trade off with our approach is that we can no longer enforce correctness in the same way as conventional lenses: our interface does not rule out all non-round-tripping BXs. We tackle this issue by proposing a new compositional reasoning framework that is flexible enough to characterise a variety of round-tripping properties, and simplifies the necessary reasoning.

Specifically, we make the following contributions:

- We describe a method to enable *monadic composition* for bidirectional programs (Section 3). Our approach is based on a construction which generates a *monadic profunctor*, parameterised on two application-specific monads which are used to generate the forward (`Fwd`) and backward (`Bwd`) directions.
- To demonstrate the flexibility of our approach, we apply the above method to three different problem domains: parsers/printers (Section 3 and 4), lenses (Section 5), and generators/predicates for structured data (Section 6). While the first two are well-explored areas in the bidirectional programming literature, the third one is a completely new application domain.
- We present a scalable reasoning framework, capturing notions of *compositionality* for bidirectional properties (Section 4). We define classes of round-tripping properties inherent to bidirectionalism, which can be verified by fol-

lowing simple criteria. These principles are demonstrated with our three examples. We include some proofs for illustration in the paper. The supplementary material [**?**] contains machine-checked Coq proofs for the main theorems.
– We have implemented these ideas as Haskell libraries [**?**], with two wrappers around attoparsec for parsers and printers, and QuickCheck for generators and predicates, showing the viability of our approach for real programs.

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

1.1 Background

We introduced lenses briefly above. We now introduce the other two bidirectional examples used in 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 have 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 actually be left- or right- inverses to each other. Other characterisations are also possible, e.g., with equivalence classes of ASTs (accounting for reassociations). Alternatively, parsers and printers may satisfy properties about the interaction of partially-parsed inputs with the printer and parser, e.g., if `parser :: String → (AST, String)`:

```
(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 [**? ? ?**].

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 extract both from a common specification with bidirectional programming is a very attractive alternative.

Roadmap We begin by outlining a concrete example of parsers and printers (Section 2), before explaining the general approach of using *monadic profunctors* to structure bidirectional programs (Section 3). Section 4 then presents a compositional reasoning framework for monadic bidirectional programs, with varying degrees of strength adapted to different round-tripping properties. We then replay the developments of the earlier sections to define lenses as well as generators and predicates in Sections 5 and 6.

2 Monadic bidirectional programming

A bidirectional parser, or *biparser*, combines both a parsing direction and printing direction. Our first novelty here is to express biparsers monadically.

In code samples, we use the functional programming pun of naming variables after their types, e.g., a variable of some polymorphic type `v` will also be called `v`. Similarly, for some parametric type `m`, a variable of type `m v` will be called `mv` and a function `u → m v` (a Kleisli arrow for a monad `m`) will be called `kv`.

Monadic parsers The following data type provides the standard way to describe parsers of values of type `v` which may consume only part of the input string:

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

It is well-known that such parsers are monadic [**?**], i.e., they have a notion of monadic sequential composition embodied by the interface:

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

The sequential composition operator `(>>=)`, called *bind*, describes the scheme of constructing a parser by sequentially composing two sub-parsers where the second depends on the output of the first; a parser of `w` values is made up of a parser of `v` and a parser of `w` that depends on the previously parsed `v`. Indeed, this is the implementation given to the monadic interface:

```
pv >>= kw = Parser (λs → let (v, s') = parse pv s in parse (kw v) s')
return v  = Parser (λs → (v, s))
```

Bind first runs the parser `pv` on an input string `s`, resulting in a value `v` which is used to create the parser `kw v`, which is in turn run on the remaining input `s'` to produce parsed values of type `w`. The `return` operation creates a trivial parser for any value `v` which does not consume any input but simply returns `v`.

Usually two monadically composed parsers have a relationship between the first parser's type and the second parser's type, and that relationship is usually containment of the former inside the latter. For example, we might parse an expression and compose this with a parser for statements, where statements contain expressions. This relationship will be useful later when we consider printers.

As a shorthand, we can discard the remaining unparsed string of a parser using projection, giving a helper function typed `parser :: Parser v → (String → v)`.

Monadic printers Our goal is to augment parsers with their inverse printer, such that we have a monadic type `Biparser` which provides two complementary (bi-directional) transformations:

```
parser  :: Biparser v → (String → v)
printer :: Biparser v → (v → String)
```

However, this type of printer `v → String` (shown also in the introduction) cannot form a monad because it is *contravariant* in its type parameter `v`. Concretely, we cannot implement the bind `(>>=)` operator for values with types of this form:

```
-- Failed attempt
bind :: (v → String) → (v → (w → String)) → (w → String)
bind pv kw = λw → let v = (??) in pv v ++ kw v w
```

We are stuck trying to fill the hole (??) as there is no way to get a value of type v to pass as an argument to pv (first printer) and kw (second printer which depends on a v). Subsequently, we cannot leverage the result that the product of two monads is a monad by taking a product of the parser monad and v → String.

But what if our types were related by containment, such that v is contained within w and thus we have a projection w → v? We could use this projection to fill the hole in the failed attempt above, defining a bind-like operator:

```
bind' :: (w → v) → (v → String) → (v → (w → String)) → (w → String)
bind' from pv kw = λw → let v = from w in pv v ++ kw v w
```

This is closer to the monadic form, where from :: w → v resolves the difficulty of contravariance by "contextualizing" the printers. Thus, the first printer is no longer just "a printer of v", but "a printer of v extracted from w". In the context of constructing a bidirectional parser, having such a function to hand is not an unrealistic expectation: recall that when we compose two parsers, typically the values of the first parser for v are contained within the values returned by the second parser for w, thus a notion of projection can be defined and used here to recover a v in order to build the corresponding printer compositionally.

Of course, this is still not a monad. However, it suggests a way to generate a monadic form by putting the printer and contextualizing projection together, (w → v, v → String) and fusing them into (w → (v, String)). This has the advantage of removing the contravariant occurence of v, yielding a data type:

```
data Printer w v = Printer { runPrinter :: w → (v, String) }
```

If we fix the first parameter type w, then the type Printer w of printers for w values is indeed monadic, combining a *reader monad* (for some global read-only parameter of type w) and a *writer monad* (for strings), with implementation:

```
instance Monad (Printer w) where
  return :: v → Printer w v
  return = λv → Printer (λ_ → (v, ""))

  (>>=) :: Printer w v → (v → Printer w t) → Printer w t
  pv >>= kt = Printer (λw → let (v, s) = print pv w
                                (t, s') = print (kt v) w in (t, s ++ s'))
```

The printer return v ignores its input and prints nothing. For bind, an input w is shared by both printers and the resulting strings are concatenated.

We can adapt the contextualisation of a printer by the following operation which amounts to pre-composition, witnessing the fact that Printer is a contravariant functor in its first parameter:

```
comap :: (w → w') → Printer w' v → Printer w v
comap from (Printer f) = Printer (f ∘ from)
```

2.1 Monadic biparsers

So far so good: we now have a monadic notion of printers. However, our goal is to combine parsers and printers in a single type. Since we have two monads, we use the standard result that a product of monads is a monad, defining *biparsers*:

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

By pairing parsers and printers we have to unify their covariant parameters. When both type parameters are the same it is easy to interpret this type: a biparser `Biparser v v` is a parser from strings to `v` values and printer from `v` values to strings. We refer to biparsers of this type as *aligned* biparsers. What about when the type parameters differ? A biparser of type `Biparser u v` provides a parser from strings to `v` values and a printer from `u` values to strings, but where the printers can compute `v` values from `u` values, i.e., `u` is some common broader representation which contains relevant `v`-typed subcomponents. A biparser `Biparser u v` can be thought of as printing a certain subtree `v` from the broader representation of a syntax tree `u`.

The corresponding monad for `Biparser` is the product of the previous two monad definitions for `Parser` and `Printer`, allowing both to be composed sequentially at the same time. To avoid duplication we elide the definition here which is shown in full in Appendix A.

We can also lift the previous notion of `comap` from printers to biparsers, which gives us a way to localize (contextualize) a printer:

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

upon :: Biparser u' v → (u → u') → Biparser u v
upon = flip comap
```

In the rest of this section, we use the alias `upon` for `comap` with flipped parameters where we read `p ‘upon‘ subpart` as applying the printer of `p :: Biparser u' v` on a subpart of an input of type `u` calculated by `subpart :: u → u'`, thus yielding a biparser of type `Biparser u v`.

An example biparser Let us write a biparser, `string :: Biparser String String`, for strings which are prefixed by their length and a space. 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 biparser of single characters as:

```
char :: Biparser Char Char
char = Biparser (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 (a singleton list) paired with `c`.

Next, we define a biparser `int` for an integer followed by a single space. An auxiliary biparser `digits` (on the right) parses an integer one digit at a time into a string. Note that in Haskell, the **do**-notation statement "`d ← char ‘upon‘ head`" desugars to "`char ‘upon‘ head >>= λ d → ...`" which uses `(>>=)` and a function binding `d` in the scope of the rest of the desugared block.

```
int :: Biparser Int Int
int = do
  ds ← digits ‘upon‘ printedInt
  return (read ds)
  where
    printedInt n = show n ++ "␣"
```

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

On the right, `digits` extracts a `String` consisting of digits followed by a single space. As a parser, it parses a character (`char ‘upon‘ head`); if it is a digit then it continues parsing recursively (`digits ‘upon‘ tail`) appending the first character to the result (`d : igits`). Otherwise, if the parsed character is a space the parser returns `"␣"`. As a printer, it expects a string of the same format, which must be non-empty; `‘upon‘ head` extracts the first character of the input, then `char` prints it and returns it back as `d`; if it is a digit, then `‘upon‘ tail` extracts the rest of the input to print recursively. If the character is a space, the printer returns a space and terminates; otherwise (not digit or space) the printer throws an error.

On the left, the biparser `int` uses `read` to convert an input string of digits parsed by `digits` to an integer, and `printedInt` to convert an integer to an output string printed by `digits`. A safer implementation could return the `Maybe` type when parsing but we keep things simple here for now.

After parsing an integer `n`, we can parse the string following it by iterating `n` times the biparser `char`. This is captured by the `replicateBiparser` combinator below, defined recursively like `digits` but with the termination condition given by an external parameter. To iterate `n` times a biparser `pv`: if `n == 0`, there is nothing to do and we return the empty list; otherwise for `n > 0`, we run `pv` once to get the head `v`, and recursively iterate `n-1` times to get the tail `vs`.

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

```
replicateBiparser :: Int → Biparser u v → Biparser [u] [v]
replicateBiparser 0 pv = return []
replicateBiparser n pv = do
  v  ← pv ‘upon‘ head
  vs ← (replicateBiparser (n - 1) pv) ‘upon‘ tail
  return (v : vs)
```

(akin to `replicateM` from Haskell's standard library). We can now fulfil our task:

```
string :: Biparser String String
string = int ‘upon‘ length >>= λn → replicateBiparser n char
```

Interestingly, if we erase applications of `upon`, i.e., we substitute every expression of the form `py `upon` f` with `py` and ignore the second parameter of the types, we obtain what is essentially the definition of a parser in an idiomatic style for monadic parsing. This is because `` `upon` f `` is the identity on the parser component of `Biparser`. Thus the biparser code closely resembles standard, idiomatic monadic parser code but with "annotations" via `upon` expressing how to apply the backwards direction of printing to subparts of the parsed string.

Despite its simplicity, the syntax of length-prefixed strings is notably context-sensitive. Thus the example makes crucial use of the monadic interface for bidirectional programming: a value (the length) must first be extracted to dynamically delimit the string to be parsed next. Context-sensitivity is standard for parser combinators in contrast with parser generators, e.g., Yacc, and applicative parsers, which are mostly restricted to context-free languages. By our monadic BX approach, we can now bring this power to bear on *bidirectional* parsing.

3 A unifying structure: monadic profunctors

The biparser examples of the last section were enabled by both the monadic structure of `Biparser` and the `comap` operation (also called `upon`, with flipped arguments). We call types which have both a monadic structure and a `comap` operation a *monadic profunctor*. The notion of a monadic profunctor is general, but it characterises a key class of structures for bidirectional programs, which we explain here. Furthermore, we show a construction of monadic profunctors from pairs of monads which elicits the necessary structure for monadic bidirectional programming in the style of the previous section.

Profunctors In Section 2.1, biparsers were defined by a data type with two type parameters (`Biparser u v`) which is functorial and monadic in the second parameter and *contravariantly* functorial in the first parameter (provided by the `comap` operation). In standard terminology, a two-parameter type `p` which is functorial in both its type parameters is called a *bifunctor*. 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] This differs slightly to the standard category theory terminology where a profunctor is a bifunctor $\mathsf{F} : \mathcal{D}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$. The Haskell community's use of the term "profunctor" makes sense if we treat Haskell in an idealised way as the category of sets.

We adopt this programming-oriented terminology here, capturing the `comap` operation via a class `Profunctor`. In the preceding section, some uses of `comap` involved a partial function, e.g., `comap head`. We make the possibility of partiality explicit via the `Maybe` type, yielding the following definition.

Definition 1. A binary data type is a **profunctor** if it is a contravariant functor in its first parameter and covariant functor in its second, with the operation:

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

```
class ForallF Functor p ⇒ Profunctor p where
    comap :: (u → Maybe u') → p u' v → p u v
```

which should obey two laws:

```
comap Just = id      comap (f >=> g) = comap f ∘ comap g
```

where `(>=>)` is the (left-to-right) composition operator for Kleisli arrows of the `Maybe` monad, i.e., composition of partial functions: `(>=>) :: (a → Maybe b) → (b → Maybe c) → a → Maybe c`.

The constraint `ForallF Functor p` captures a universally quantified constraint [**?**]: for all types `u` then `p u` has an instance of the `Functor` class.[5]

Since the contravariant part of the bifunctor applies to functions of type `u → Maybe u'`, the categorical analogy here is more precisely a profunctor $\mathsf{F} : {\mathcal{C}_T}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ where $\mathcal{C}_T$ is the Kleisli category of the partiality (`Maybe`) monad.

The need for `comap` to take partial functions is in response to the frequent need to restrict the domain of bidirectional transformations. In combinator-based approaches, the combinators typically constrain bidirectional programs to be bijections, enforcing domain restrictions by construction. Our more flexible approach means we need a way to include such restrictions explicitly– hence `comap`.

Definition 2. A **monadic profunctor** is a profunctor `p` (in sense of Definition 1) such that `p u` is a monad for all `u`. In terms of type class constraints, this means there is an instance `Profunctor p` and for all `u` there is a `Monad (p u)` instance. Thus, we represent monadic profunctors by the following empty class (which inherits all its methods from its superclasses):

```
class (Profunctor p, ForallF Monad p) ⇒ Profmonad p
```

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 (partial) functions into monad morphisms. In Haskell, these laws are obtained *for free* by parametricity [**?**]. This means that every contravariant functor and monad is in fact a monadic profunctor, thus the following universal instance is lawful:

```
instance (Profunctor p, ForallF Monad p) ⇒ Profmonad p
```

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

Lastly, we introduce a useful piece of terminology (mentioned in the previous section on biparsers) for describing values of a profunctor of a particular form:

Definition 3. A value `p :: P u v` of a profunctor `P` is called *aligned* if `u` = `v`.

[5] GHC will shortly allow universal quantification in its constraints, written as `forall u . Functor (p u)`, but until then we use the constraint constructor `ForallF` from the `constraints` package: `http://hackage.haskell.org/package/constraints`.

3.1 Constructing monadic profunctors

Our examples (parsers/printers, lenses, and generators/predicates) share monadic profunctors as an abstraction, making it possible to write different kinds of bidirectional transformations monadically. Underlying these definitions of monadic profunctors is a common structure, which we explain here using biparsers, and which will be replayed in Section 5 for lenses and Section 6 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`. These are standard mathematical constructions, and the latter appears in the Haskell profunctors package with the name `Star`. Our core construction is based on these two ways of creating a profunctor, which we call `Fwd` and `Bwd` respectively:

```
data Fwd m u v = Fwd { unFwd :: m v }      -- ignore contrv. parameter
data Bwd m u v = Bwd { unBwd :: u → m v } -- maps from contrv. parameter
```

The naming reflects the idea that these two constructions will together capture a bidirectional transformation and are related by domain-specific round-tripping properties in our framework. Both `Fwd` and `Bwd` map any functor into a profunctor by the following type class instances:

```
instance Functor m ⇒ Functor (Fwd m u) where
  fmap f (Fwd x) = Fwd (fmap f x)
instance Functor m ⇒ Profunctor (Fwd m) where
  comap f (Fwd x) = Fwd x

instance Functor m ⇒ Functor (Bwd m u) where
  fmap f (Bwd x) = Bwd ((fmap f) ∘ x)
instance (Monad m, MonadPartial m) ⇒ Profunctor (Bwd m) where
  comap f (Bwd x) = Bwd ((toFailure ∘ f) >=> x)
```

There is an additional constraint here for `Bwd`, enforcing that the monad `m` is a member of the `MonadPartial` class which we define as:

```
class MonadPartial m  where  toFailure :: Maybe a → m a
```

This provides an interface for monads which can internalise a notion of failure, as captured at the top-level by `Maybe` in `comap`.

Furthermore, `Fwd` and `Bwd` both map any monad into a monadic profunctor:

```
instance Monad m                    instance Monad m
      ⇒ Monad (Fwd m u) where             ⇒ Monad (Bwd m u) where
  return x = Fwd (return x)           return x = Bwd (λ_ → return x)
  Fwd py >>= kz =                     Bwd my >>= kz = Bwd
    Fwd (py >>= unFwd ∘ kz)             (λu → my u >>= (λy → unBwd (kz y) u))
```

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 (ForallF Functor (p :*: q), Profunctor p, Profunctor q)
      ⇒ Profunctor (p :*: q) where
  comap f (py :*: qy) = comap f py :*: comap f qy
```

3.2 Deriving biparsers as monadic profunctor pairs

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

```
type State  s a = s → (a, s)
type WriterT w m a = m (a, w)
type Biparser = Fwd (State String) :*: Bwd (WriterT Maybe String)
```

The backward direction composes the writer monad with the `Maybe` monad using `WriterT` (the writer monad transformer, equivalent to composing two monads with a distributive law). Thus the backwards component of `Biparser` corresponds printers (which may fail) and the forwards component to parsers:

```
Bwd (WriterT Maybe String) u v   ≅   u → Maybe (v, String)
        Fwd (State String) u v   ≅   String → (v, String)
```

For the above code to work in Haskell, the `State` and `WriterT` types need to be defined via either a **`data`** type or **`newtype`** in order to allow type class instances on partially applied type constructors. We abuse the notation here for simplicity but define smart constructors and deconstructors for the actual implementation:[6]

```
parse :: Biparser u v → (String → (v, String))
print :: Biparser u v → (u → Maybe (v, String))
mkBiparser :: (String → (v, String)) → (u → Maybe (v, String)) → Biparser u v
```

The monadic profunctor definition for biparsers now comes for free from the constructions in Section 3.1 along with the following instance of `MonadPartial` for the writer monad transformer with the `Maybe` monad:

```
instance Monoid w ⇒ MonadPartial (WriterT w Maybe) where
  toFailure Nothing = WriterT Nothing
  toFailure (Just a) = WriterT (Just (a, mempty))
```

In a similar manner, we will use this monadic profunctor construction to define monadic bidirectional transformations for lenses (§5) and bigenerators (§6).

The example biparsers from Section 2.1 can be easily redefined using the structure here. For example, the primitive biparser `char` becomes:

```
char :: Biparser Char Char
char = mkBiparser (λ (c : s) → (c, s)) (λ c → Just (c, [c]))
```

[6] *Smart constructors* (and dually *smart deconstructors*) are just functions that hide boilerplate code for constructing and deconstructor data types.

Codec library The codec library [**?**] provides a general type for bidirectional programming isomorphic to our composite type `Fwd r :*: Bwd 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 with codecs.

4 Reasoning about bidirectionality

So far we have seen how the monadic profunctor structure provides a way to define biparsers using familiar operations and syntax: monads and **do**-notation. This structuring allows both the forwards and backwards components of a biparser to be defined simultaneously in a single compact definition.

This section studies the interaction of monadic profunctors with the *round-tripping laws* that relate the two components of a bidirectional program. For every bidirectional transformation we can define dual properties: *backward round tripping* (going backwards-then-forwards) and *forward round tripping* (going forwards-then-backwards). In each BX domain, such properties will also capture the domain-specific information flow inherent to the transformations. We use biparsers as the running example. We then apply the same principles to our other examples in Sections 5 and 6. For brevity, we use `Bp` as an alias for `Biparser`.

Definition 4. A biparser `p :: Bp u u` is *backward round tripping* if for all `x :: u` and `s, s' :: String` then: (recall `print p :: u → Maybe (v, String)`)

```
fmap snd (print p x) = Just s   ⟹   parse p (s ++ s') = (x, s').
```

That is, if a biparser `p` when used as a printer (going backwards) on an input value `x` produces a string `s`, then using `p` as a parser on a string with prefix `s` and suffix `s'` yields the original input value `x` and the remaining input `s'`.

Note that backward round tripping is defined for *aligned* biparsers (of type `Bp u u`) since the same value `x` is used as both the input of the printer (typed by the first type parameter of `Bp`) and as the expected output of the parser (typed by the second type parameter of `Bp`).

The dual property is *forward* round tripping: a source string `s` is parsed (going forwards) into some value `x` which when printed produces the initial source `s`:

Definition 5. A biparser `p :: Bp u u` is *forward round tripping* if for every `x :: u` and `s :: String` we have that:

```
parse p s = (x, "")   ⟹   fmap snd (print p x) = Just s
```

Proposition 1 The biparser `char :: Bp Char Char` (§3.2) is both backward and forward round tripping. Proof by expanding definitions and algebraic reasoning.

Note, in some applications, forward round tripping is too strong. Here it requires that every printed value corresponds to at most one source string. This is often not the case as ASTs typically discard formatting and comments so that pretty-printed code is lexically different to the original source. However, different notions of equality enable more reasonable forward round-tripping properties.

Although one can check round-tripping properties of biparsers by expanding their definitions and the underlying monadic profunctor operations, a more scalable approach is provided if a round-tripping property is *compositional* with respect to the monadic profunctor operations, i.e., if these operations preserve the property. Compositional properties are easier to enforce and check since only the individual atomic components need round-tripping proofs. Such properties are then guaranteed "by construction" for programs using those components.

4.1 Compositional properties of monadic bidirectional programming

Let us first formalize compositionality as follows. A *property* $\mathcal{R}$ over a monadic profunctor `P` is a family of subsets $\mathcal{R}_{\mathtt{v}}^{\mathtt{u}}$ of `P u v` indexed by types `u` and `v`.

Definition 6. A property $\mathcal{R}$ over a monadic profunctor `P` is *compositional* if the monadic profunctor operations are closed over $\mathcal{R}$, i.e., the following conditions hold for all types `u`, `v`, `w`:

1. For all `x :: v`, $\mathtt{(return\ x)} \in \mathcal{R}_{\mathtt{v}}^{\mathtt{u}}$ (comp-return)
2. For all `p :: P u v` and `k :: v → P u w`,
$$\left(\mathtt{p} \in \mathcal{R}_{\mathtt{v}}^{\mathtt{u}}\right) \wedge \left(\forall \mathtt{v}.\ \mathtt{(k\ v)} \in \mathcal{R}_{\mathtt{w}}^{\mathtt{u}}\right) \implies \mathtt{(p >>= k)} \in \mathcal{R}_{\mathtt{w}}^{\mathtt{u}} \quad \text{(comp-bind)}$$
3. For all `p :: P u' v` and `f :: u → Maybe u'`,
$$\mathtt{p} \in \mathcal{R}_{\mathtt{v}}^{\mathtt{u'}} \implies \mathtt{(comap\ f\ p)} \in \mathcal{R}_{\mathtt{v}}^{\mathtt{u}} \quad \text{(comp-comap)}$$

Unfortunately for biparsers, forward and backward round tripping as defined above are *not* compositional: `return` is not backward round tripping and `>>=` does not preserve forward round tripping. Furthermore, these two properties are restricted to biparsers of type `Bp u u` (i.e., aligned biparsers) but compositionality requires that the two type parameters of monadic profunctor can differ in the case of `comap` and `(>>=)`. This suggests that we need to look for more general properties that capture the full gamet of possible biparsers.

We first focus on backward round tripping. Informally, backward round tripping states that if you print (going backwards) and parse the resulting output (going forwards) then you get back the initial value. However, in a general biparser `p :: Bp u v`, the input type of the printer `u` differs from the output type of the parser `v`, so we cannot compare them. But our intent for printers is that what we actually print is a fragment of `u`, a fragment which is given as the output of the printer. By thus comparing the outputs of both the parser and printer, we obtain the following variant of backward round tripping:

Definition 7. A biparser `p :: Bp u v` is *weak backward round tripping* if for all `x :: u`, `y :: v`, and `s`, `s' :: String` then:

```
print p x = Just (y, s)   ⟹   parse p (s ++ s') = (y, s')
```

Removing backward round-tripping's restriction to aligned biparsers and using the result `y :: v` of the printer gives us a property that *is* compositional:

Proposition 2 Weak backward round tripping of biparsers is compositional.

Proposition 3. The primitive biparser `char` is weak backward round tripping.

Corollary 2. Propositions 2 & 3 imply `string` is weak backward round tripping.

This property is "weak" as it does not constrain the relationship between the input `u` of the printer and its output `v`. In fact, there is no hope for a compositional property to do so: the monadic profunctor combinators do not enforce a relationship between them. However, we can regain compositionality for the stronger backward round-tripping property by combining the weak compositional property with an additional non-compositional property on the relationship between the printer's input and output. This relationship is represented by the function that results from ignoring the printed string, which amounts to removing the main effect of the printer. Thus we call this operation a *purification*:

```
purify :: forall u v. Bp u v → u → Maybe v
purify p u = fmap fst (print p u)
```

Ultimately, when a biparser is aligned (`p :: Bp u u`) we want an input to the printer to be returned in its output, i.e, `purify p` should equal `λx → Just x`. If this is the case, we recover the original backward round tripping property:

Theorem 1 If `p :: P u u` is weak backward round tripping, and for all `x :: u`. `purify p x = Just x`, then `p` is backward round tripping.

Thus, for any biparser `p`, we can get backward round tripping by proving that its atomic subcomponents are weak backward round tripping, and proving that `purify p x = Just x`. The interesting aspect of the purification condition here is that it renders irrelevant the domain-specific effects of the biparser, i.e., those related to manipulating source strings. This considerably simplifies any proof. Furthermore, the definition of `purify` is a *monadic profunctor homomorphism* which provides a set of equations that can be used to expedite the reasoning.

Definition 8. A *monadic profunctor homomorphism* between monadic profunctors `P` and `Q` is a polymorphic function `proj :: P u v → Q u v` such that:

$$
\begin{aligned}
\texttt{proj (comap}_P\ \texttt{f p)} &\equiv \texttt{comap}_Q\ \texttt{f (proj p)} \\
\texttt{proj (p >>=}_P\ \texttt{k)} &\equiv \texttt{(proj p) >>=}_Q\ \texttt{(λx → proj (k x))} \\
\texttt{proj (return}_P\ \texttt{x)} &\equiv \texttt{return}_Q\ \texttt{x}
\end{aligned}
$$

Proposition 4. The `purify :: Bp u v → u → Maybe v` operation for biparsers is a monadic profunctor homomorphism between `Bp` and the monadic profunctor `PartialFun u v = u → Maybe v`.

Corollary 3. (of Theorem 1 with Corollary 2 and Proposition 4) The biparser `string` is backward round tripping.

Proof. First prove (in Appendix B) the following properties of biparsers `char`, `int`, and `replicatedBp :: Int → Bp u v → Bp [u] [v]` (writing `proj` for `purify`):

$$\texttt{proj char n} \equiv \texttt{Just n} \tag{4.1}$$

$$\texttt{proj int n} \equiv \texttt{Just n} \tag{4.2}$$

$$\texttt{proj (replicateBp (length xs) p) xs} \equiv \texttt{mapM (proj p) xs} \tag{4.3}$$

From these and the homomorphism properties we can prove `proj string = Just`:

$$\begin{aligned}
&\texttt{proj string xs} \\
\equiv\ &\texttt{proj (comap length int >>= λn → replicateBp n char) xs} \\
\textit{Prop.4} \equiv\ &\texttt{(comap length (proj int) >>= λn → proj (replicateBp n char)) xs} \\
(4.2) \equiv\ &\texttt{(comap length Just >>= λn → proj (replicateBp n char)) xs} \\
\textit{Def.2} \equiv\ &\texttt{proj (replicateBp (length xs) char) xs} \\
(4.3) \equiv\ &\texttt{mapM (proj char) xs} \\
(4.1) \equiv\ &\texttt{mapM Just xs} \\
\{\textit{monad}\} \equiv\ &\texttt{Just xs}
\end{aligned}$$

Combining `proj string = Just` with Corollary 2 (`string` is weak backward round tripping) enables Theorem 1, giving that `string` is backward round tripping.

The other two core examples in this paper also permit a definition of `purify`. We capture the general pattern as follows:

Definition 9. A *purifiable monadic profunctor* `P` is a monadic profunctor equipped with a homomorphism `proj` from `P` to the partial functions monadic profunctor `- → Maybe -`. We say that `proj p` is the *pure projection* of `p`.

Definition 10. A pure projection `proj p :: u → Maybe v` is called the *identity projection* when `proj p x = Just x` for all `x :: u`.

Here and in Sections 5 and 6, identity projections enable compositional round-tripping properties to be derived from more general non-compositional properties, as seen above for backward round tripping of biparsers.

We have neglected forward round tripping, which is not compositional, not even in an a weakened form. However, we can generalize compositionality with conditions related to *injectivity*, enabling a generalisation of forward round tripping. We call the generalized meta-property *quasicompositionality*.

4.2 Quasicompositionality for monadic profunctors

An injective function $f : A \to B$ is a function for which there exists a left inverse $f^{-1} : B \to A$, i.e., where $f^{-1} \circ f = id$. We can see this pair of functions as a simple kind of bidirectional program, with a forward round-tripping property (assuming f is the forwards direction). We can lift the notion of injectivity to the monadic profunctor setting and capture forward round-tripping properties that are preserved by the monadic profunctor operations, given some additional injectivity-like restriction. We first formalise the notion of an *injective arrow*:

Definition 11. Let `m` be a monad. A function `k :: v → m w` is an *injective arrow* if there exists `k' :: w → v` (called the *left arrow inverse* of `k`) and for all `x :: v`:

$$\texttt{k x >>= λy → return (x, y)} \quad \equiv \quad \texttt{k x >>= λy → return (k' y, y)}$$

Informally, an injective arrow represents a computation producing an output `y` from which the "header" `x` that is an input to the arrow can be extracted back.

Next, we define *quasicompositionality* which extends the compositionality meta-property with the requirement for `>>=` to be applied to injective arrows:

Definition 12. Let `P` be a monadic profunctor. A property $\mathcal{R}^{\texttt{u}}_{\texttt{v}} \subseteq \texttt{P u v}$ indexed by types `u` and `v` is *quasicompositional* if the following holds

1. For all `x :: v`, $\texttt{(return x)} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}$ (qcomp-return)
2. For all `p :: P u v, k :: v → P u w`, if `k` is an injective arrow,
$$\texttt{p} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}} \wedge (\forall \texttt{v}.\, \texttt{k v} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}}) \Longrightarrow \texttt{(p >>= k)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}} \qquad \text{(qcomp-bind)}$$
3. For all `p :: P u' v, f :: u → Maybe u'`,
$$\texttt{p} \in \mathcal{R}^{\texttt{u'}}_{\texttt{v}} \wedge \Longrightarrow \texttt{(comap f p)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}} \qquad \text{(qcomp-bind)}$$

We now formulate a weakening of forward round tripping. As with weak backward round tripping, we rely on the idea that the printer *outputs* both a string and the value that was printed, so that we need to compare the outputs of both the parser and the printer, as opposed to comparing the output of the parser with the input of the printer as in (strong) forward round tripping. If running the parser component of a biparser on a string `s01` yields a value `y` and a remaining string `s1`, and the printer ever outputs that same value `y` along with a string `s0`, then `s0` is the prefix of `s01` that was consumed by the biparser: `s01 = s0 ++ s1`.

Definition 13. A biparser `p : Bp u v` is *weak forward round tripping* if for all `x :: u`, `y :: v`, and `s0, s1, s01 :: String` then:

$$\texttt{parse p s01 = (y, s1)} \ \wedge \ \texttt{print p x = Just (y, s0)} \ \Longrightarrow \ \texttt{s01 = s0 ++ s1}$$

Proposition 5. Weak forward round tripping is quasicompositional.

Proof. We sketch the qcomp-bind case, where `p = (m >>= k)` for some `m` and `k` that are weak forward roundtripping. From `parse (m >>= k) s01 = (y, s1)`, it follows that there exists `(z, s)` such that `parse m s01 = (z, s)` and `parse (k z) s = (y, s1)`. Similarly `print (m >>= k) x = Just (y, s')` implies there exist `z'`, `s0'` such that `print m x = Just (z', s0')` and `print (k z') x = Just (y, s1')` and `s = s0' ++ s1'`. Because `k` is an injective arrow, we have `z = z'`. Thus we can use the assumption that `m` and `k` are weak forward roundtripping once on `m` and once on `k a`, and deduce that `s01 = s0' ++ s` and `s = s1' ++ s1`. Conclude with a bit of algebra.

Proposition 6. The `char` biparser is weak forward round tripping.

Corollary 4. Propositions 5 and 6 imply that `string` is weak forward round tripping if we restrict the parser to inputs whose digits do not contain redundant leading zeros.

Proof. All of the right operands of `>>=` in the definition of `string` are injective arrows, apart from `λds → return (read ds)` at the end of the auxiliary `int` biparser. Indeed, the `read` function is not injective since multiple strings may parse to the same integer: `read "0" = read "00" = 0`. But the pre-condition to the proposition (no redundant leading zero digits) restricts the input strings so that `read` is injective. The rest of the proof is a simple corollary of Propositions 5 and 6.

Thus, quasicompositionality gives us scalable reasoning for weak forward round tripping, which is by construction for biparsers: we just need to prove this property for individual atomic biparsers. Similarly to backward round tripping, we can prove forward round tripping by combining weak forward round tripping with the identity projection property:

Theorem 2. If `p :: P u u` is weak forward round-tripping, and for all `x :: u`, `purify p x = Just x`, then `p` is forward round tripping.

Corollary 5. The biparser `string` is forward round tripping by the above theorem (with identity projection shown in the proof of Corollary 3) and Corollary 4.

Summary Whilst combinator-based approaches to BX can guarantee round-tripping by construction, we have made a trade-off to get greater expressivity in the monadic approach. However, we've shown here how to regain the ability to reason about bidirectional transformations in a manageable way.

For any BX we can consider two round-tripping properties: fowards-then-backwards and backwards-then-forwards, called just *forward* and *backward* here respectively. If such properties are compositional, then we have scalable reasoning. But due to the monadic profunctor structuring this tends not to be the case. Instead, there is a weakening which is compositional or quasicompositional (adding injectivity). In such cases, we recover the stronger property by proving a simple property on aligned transformations: that the backwards direction faithfully reproduces its input as its output (*identity projection*). We apply this approach in the next two sections, but in less detail.

5 Monadic bidirectional programming for lenses

Lenses are a common object of study in bidirectional programming, comprising a pair of functions ($\texttt{get} : S \to V, \texttt{put} : V \to S \to S$) satisfying *well-behaved lens* laws shown in Section 1. Previously, when considering the monadic structure of parsers and printers, the starting point was that parsers already have a well-known monadic structure. The challenge came in finding a reasonable monadic characterisation for printers that was compatible with the parser monad. In the end, this construction was expressed by a product of two monadic profunctors `Fwd m` and `Bwd n` for monads `m` and `n`. For lenses we are in the same position: the forwards direction (`get`) is already a monad—the reader monad. The backwards direction `put` is not a monad since it is contravariant in its parameter; the same situation as printers. We can apply the same approach of "monadisation" used for parsers and printers, giving the following new data type for lenses:

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

Thus, the result of the backward direction is paired with a covariant parameter `v` in the same way as monadic printers. Instead of mapping a view and a source to a source, `put` now maps values of a different type `u`, which we call a *pre-view*, along with a source `s` into a pair of a view `v` and source `s`. This definition can be structured as a monadic profunctor via a pair of `Fwd` and `Bwd` constructions:

```
type L s = (Fwd (Reader s)) :*: (Bwd (State s))
```

Thus by the results of Section 3, we now have a monadic profunctor characterisation of lenses that allows us to compose lenses via the monadic interface.

Ideally, `get` and `put` should be total, but this is impossible without a way to restrict the domains. In particular, there is the known problem of "duplication" [**?**], where source data may appear more than once in the view, and a necessary condition for `put` to be well-behaved is that the duplicates remain equal amid view updates. This problem is inherent to all bidirectional transformations, and bidirectional languages have to rule out inconsistent updates of duplicates either statically [**?**] or dynamically [**?**]. To remedy this, we capture both partiality of `get` and a predicate on sources in `put` for additional dynamic checking. This is provided by the following `Fwd` and `Bwd` monadic profunctors:

```
type ReaderT r m a = r → m a
type StateT s m a  = s → m (a, s)
type WriterT w m a = m (a, w)

type L s = (Fwd (ReaderT s Maybe))
       :*: (Bwd (StateT s (WriterT (s → Bool) Maybe)))

-- Smart deconstructors:
get :: L s u v → (s → Maybe v)
put :: L s u v → (u → s → Maybe ((v, s), s → Bool))
```

Going forwards, *getting* a view `v` from a source `s` may fail if there is no view for the current source. Going backwards, *putting* a pre-view `u` updates some source `s` (via the state transformer `StateT s`), but with some further structure returned, provided by `WriterT (s → Bool) Maybe` (similar to the writer transformer used for biparsers, § 3.2, p. 12). The `Maybe` here captures the possibility that `put` can fail. The `WriterT (s → Bool)` structure provides a predicate which detects the "duplication" issue mentioned earlier. Informally, the predicate can be used to check that previously modified locations in the source are not modified again. For example, if a lens has a source made up of a bit vector, and a `put` sets bit i to 1, then the returned predicate will return `True` for all bits vectors where bit i is 1, and `False` otherwise. This predicate can then be used to test whether further `put` operations on the source have modified bit i.

Similarly to biparsers, a pre-view `u` can be understood as *containing* the view `v` that is to be merged with the source, and which is returned with the updated source. Ultimately, we wish to form lenses of matching input and output types (i.e. `L s v v`) satisfying the standard lens well-behavedness laws modulo explicit

management of partiality via `Maybe` and testing for conflicts via the predicate:

$$\texttt{put l x s = Just ((\_, s'), p')} \wedge \texttt{p' s'} \implies \texttt{get l s' = Just x} \quad \text{(L-PutGet)}$$

$$\texttt{get l s = Just x} \implies \texttt{put l x s = Just ((\_, s), \_)} \quad \text{(L-GetPut)}$$

L-PutGet and L-GetPut are backward and forward round tripping respectively. Some lenses, such as the later example, are not defined for all views. In that case we may say that the lens is backward/forward round tripping in some subset `P` $\subseteq$ `u` when the above properties only hold when `x` is an element of `P`.

For every source type `s`, the lens type `L s` is automatically a monadic profunctor by its definition as the pairing of `Fwd` and `Bwd` (Section 3.1), and the following instance of `MonadPartial` for handling failure and instance of `Monoid` to satisfy the requirements of the writer monad:

```
instance MonadPartial (StateT s (WriterT (s → Bool) Maybe)) where
  toFailure Nothing  = StateT (λ_ → WriterT Nothing)
  toFailure (Just x) = StateT (λs → WriterT (Just ((x , s), mempty)))

instance Monoid (s → Bool) where
  mempty      = λ_ → True
  mappend h j = λs0 → h s0 && j s0
```

A simple lens example operates on key-value maps. For keys of type `Key` and values of type `Value`, we have the following source type and a simple lens:

```
type Src = Map Key Value
atKey :: Key → L Src Value Value  -- Key-focussed lens
atKey k = mkLens (lookup k)
  (λv → λmap → Just ((v, insert k v map), λm' → lookup k m' == Just v))
```

The `get` component of the `atKey` lens does a lookup of the key `k` in a map, producing `Maybe` of a `Value`. The `put` component inserts a value for key `k`. When the key already exists, `put` overwrites its associated value.

Due to our approach, multiple calls to `atKey` can be composed monadically, giving a lens that gets/sets multiple key-value pairs at once. The list of keys and the list of values are passed separately, and are expected to be the same length.

```
atKeys :: [Key] → L Src [Value] [Value]
atKeys [] = return []
atKeys (k : ks) = do
  x  ← comap headM (atKey k)     -- headM :: [a] → Maybe a
  xs ← comap tailM (atKeys ks)   -- tailM :: [a] → Maybe [a]
  return (x : xs)
```

We refer interested readers to our implementation [**?**] for more examples, including further examples involving trees.

Round tripping We apply the reasoning framework of Section 4, taking the standard lens laws as the starting point (neither of which are compositional).

We first weaken backward round tripping to be compositional. Informally, the property expresses the idea, that if we put some value `y` in a source, resulting in

a source `z'`, then what we get from `z'` is `y`. However two important changes are needed to adapt to our generalized type of lenses and to ensure compositionality. First, the value `y` that was put is now to be found in the output of `put`, whereas there is no way to constrain the input `x` because its type `u` is abstract. Second, by sequentially composing lenses such as in `l >>= k`, the output source `z'` of `put l` will be further modified by `put (k y)`, so this round-tripping property must constrain all potential modifications of `z'`. In fact, the predicate `p` ensures exactly that the view `get l` has not changed and is still `y`. It is not even necessary to refer to `z'`, which is just one source for which we expect `p` to be `True`.

Definition 14. A lens `l :: L s u v` is *weak backward round tripping* if for all `x :: u`, `y :: v`, `z, z'' :: s`, and `p :: s → Bool`, we have:

```
put l x z = Just ((y, _), p) ∧ p z'' = True   ⟹   get l z'' = Just y
```

Theorem 3. Weak backward round tripping is a compositional property.

Again, we complement this weakened version of round tripping with the notion of purification.

Proposition 7. Our lens type `L` is a *purifiable* monadic profunctor (Definition 9), with a family of pure projections `proj s` indexed by a source `s`, defined:

```
proj :: s → L s u v → (u → Maybe v)
proj s = λl u → fmap (fst ∘ fst) (put l u s)
```

Theorem 4 If a lens `l` is weak backward round tripping and has identity projections on some subset `P` for all `s` (`P x ⇒ proj s l x = Just x`) then it is also backward round tripping on `P`.

To demonstrate, we apply this result to `atKeys :: [Key] → L Src [Value] [Value]`.

Proposition 8 Atomic lens `atKey k` is weak backward round tripping.

Proposition 9 Atomic lens `atKey k` has identity projection: `proj (atKey k) = Just`.

Our lens `atKeys ks` is therefore weak backward round tripping by construction. We now interpret/purify `atKeys ks` as a partial function, which is actually the identity function when restricted to lists of the same length as `ks`.

Proposition 10 For all `vs :: [Value]` such that `length vs = length ks`, and for all `z :: Src` then `proj z (atKeys ks) vs = Just vs`.

Corollary 6. By the above results, `atKeys ks :: L Src [Value] [Value]` for all `ks` is backward round tripping on lists of length `length ks`.

The other direction, forward round tripping, follows a similar story. We first restate it as a quasicompositional property.

Definition 15. A lens `l :: L s u v` is *weak forward round tripping* if for all `x :: u`, `y :: v`, `z, z' :: s`, and `p :: s → Bool`, we have:

```
get l z = Just y ∧ put l x z = Just ((y, z'), _)   ⟹   z = z'
```

Theorem 5. Weak forward round tripping is a quasicompositional property.

Along with identity projection, this gives the original forward L-GetPut property.

Theorem 6 If a lens `l` is weak forward round tripping and has identity projections on some subset `P` for all `s` (`P x` ⇒ `proj s l x = Just x`) then it is also forward round tripping on `P`.

We can thus apply this to our example (details omitted).

Proposition 11. For all `ks`, the lens `atKeys ks :: L Src [Value] [Value]` is forward round tripping on lists of length `length ks`.

6 Monadic bidirectional programming for generators

Lastly, we capture the novel notion of *bidirectional generators* (*bigenerators*) extending random generators in property-based testing frameworks like *QuickCheck* [**?**] to a bidirectional setting. The forwards direction generates values conforming to a specification; the backwards direction checks whether values conform to a predicate. We capture the two together via our monadic profunctor pair as:

```
type G = (Fwd Gen) :*: (Bwd Maybe)
-- ... with deconstructors and constructors
generate :: G u v → Gen v                    -- forward direction
check    :: G u v → u → Maybe v              -- backward direction
mkG      :: Gen v → (u → Maybe v) → G u v
```

The forwards direction of a bigenerator is a generator, while the backwards direction is a partial function `u → Maybe v`. A value `G u v` represents a subset of `v`, where `generate` is a generator of values in that subset and `check` maps pre-views `u` to members of the generated subset. In the backwards direction, `check g` defines a predicate on `u`, which is true if and only if `check g u` is `Just` of some value. The function `toPredicate` extracts this predicate from the backward direction:

```
toPredicate :: G u v → u → Bool
toPredicate g x = case check g x of Just _  → True; Nothing → False
```

The bigenerator type `G` is automatically a monadic profunctor due to our construction (§3). Thus, monad and profunctor instances come for free, modulo (un)wrapping of constructors and given a trivial instance of `MonadPartial`:

```
instance MonadPartial Maybe where toFailure = id
```

Due to space limitations, we refer readers to Appendix D for an example of a compositionally-defined bigenerator that produces binary search trees.

Round tripping A random generator can be interpreted as the set of values it may generate, while a predicate represents the set of values satisfying it. For a bigenerator `g`, we write `x ∈ generate g` when `x` is a possible output of the generator. The generator of a bigenerator `g` should match its predicate `toPredicate g`. This requirement equates to round-tripping properties: a bigenerator is *sound* if every

value which it can generate satisfies the predicate (forward round tripping); a bigenerator is *complete* if every value which satisfies the predicate can be generated (backward round tripping). Completeness is often more important than soundness in testing because unsound tests can be filtered out by the predicate, but completeness determines the potential adequacy of testing.

Definition 16. A bigenerator `g :: G u u` is *complete* (backward round tripping) when `toPredicate g x = True` implies `x ∈ generate g`.

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

Similarly to backward round tripping of biparsers and lenses, completeness can be split into a compositional weak completeness and a purifiable property.

As before, the compositional weakening of completeness relates the forward and backward components by their outputs, which have the same type.

Definition 18. A bigenerator `g :: G u v` is *weak-complete* when

$$\texttt{check g x = Just y} \implies \texttt{y} \in \texttt{generate g}.$$

Theorem 7. Weak completeness is compositional.

In a separate step, we connect the input of the backward direction, *i.e.*, the checker, by reasoning directly about its pure projection (via a more general form of identity projection) which is defined to be the checker itself:

Theorem 8. If a bigenerator `g :: G u u` is complete if it is weak-complete and its checker satisfies the pure projection property that:

$$\texttt{check g x = Just x'} \implies \texttt{x = x'},$$

Thus to prove completeness of a bigenerator `g :: G u u`, we first have weak-completeness by construction, and we can then show that `check g` is a restriction of the identity function, interpreting all bigenerators simply as partial functions.

Considering the other direction, soundness, there is unfortunately no decomposition into a quasicompositional property and a property on pure projections. To see why, let `bool` be a random uniform bigenerator of booleans, then consider for example, `comap isTrue bool` and `comap isTrue (return True)`, where `isTrue True = Just True` and `isTrue False = Nothing`. Both satisfy any quasicompositional property satisfied by `bool`, and both have the same pure projection `isTrue`, and yet the former is unsound—it can generate `False`, which is rejected by `isTrue`—while the latter is sound. This is not a problem in practice, as unsoundness, especially in small scale, is inconsequential in testing. But it does raise an intellectual challenge: an interesting point in the design space where ease of reasoning has given way to the greater expressivity of the monadic approach.

7 Discussion and Related Work

Bidirectional transformations are a widely applicable technique used in many domains [**?**]. Among language-based solutions, the lens framework is most influential [**?** **?** **?** **?** **?** **?**]. Broadly speaking, combinators are used as programming constructs with which complex lenses are created by combining simpler ones. The combinators preserve round tripping, and therefore the resulting programs are correct by construction. A problem with lens languages is that they tend to be disconnected from more general programming. Lenses can only be constructed by very specialised combinators and are not subject to existing abstraction mechanisms. Our approach allows bidirectional transformations to be built using standard components of functional programming, and gives a reasoning framework for studying compositionality of round-tripping properties.

The framework of *applicative lenses* [**?**] uses a function representation of lenses to lift the point-free restriction of the combinator-based languages, and enables 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-abstractions and functional applications, which is not directly related to applicative functors. In a subsequent work, the authors developed a language known as HOBiT [**?**], which went further in featuring proper binding of variables. Despite the success in supporting λ-abstractions and function applications in programming bidirectional transformations, none of the languages have explored advanced patterns such as monadic programming.

The work on *monadic lenses* [**?**] investigates lenses with effects. For instance, a "put" could require additional input to resolve conflicts. Representing effects with monads helps reformulate the laws of round-tripping. In contrast, we made the type of lenses itself a monad, and showed how they can be composed monadically. 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, reusing our reasoning framework.

Other work leverages profunctors for bidirectionality. Notably, 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`. Profuctor optics and our monadic profunctors offer orthogonal composition patterns: profunctor optics can be composed "vertically" using function composition, whereas monadic profunctor composition is "horizontal" providing sequential composition. In both cases, composition in the other direction can only be obtained by breaking the abstraction.

It is folklore in the Haskell community that profunctors can be combined with applicative functors [**?**]. The pattern is sometimes called *monoidal profunctors*. The `codec` library [**?**] mentioned in Section 3 prominently features two applications of this applicative programming style: binary serialization (a form of parsing/printing) and conversion to and from JSON structures (analogous to lenses above). Opaleye [**?**], 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.

Our framework adapts gracefully to applicative programming, 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 modification. Besides our generalization to monads, purification and (quasi)compositionality for verifying round-tripping properties are novel in our framework.

Rendel and Ostermann proposed an interface for programming parsers and printers together [**?**], but they were unable to reuse the exiting structure of `Functor`, `Applicative` and `Alternative` classes (because of the need to handle types that are both covariant and contravariant), and had to reproduce the entire hierarchy separately. In contrast, our approach reuses the standard type class hierarchy, further extending the expressive power of bidirectional programming in Haskell. FliPpr [**?** **?**] is an invertible language that generates a parser from a definition of pretty-printer. In this paper, our biparser definitions are more similar to those of parsers than printers. This makes sense as it has been established that many parsers are monadic. Similar to the case of HOBiT, there is no discussion of monadic programming in FliPpr.

Previous approaches to unifying random generators and predicates mostly focused on deriving generators from predicates. One general technique evaluates predicates lazily to drive generation (random or enumerative) [**?** **?**], but one loses control over the resulting distribution of generated values. *Luck* [**?**] is a domain-specific language blending narrowing and constraint solving to specify 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.

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 BX programming, that does not restrict programmers to stylised interfaces. Our example of generators broadened the scope of BX programming from transformations (converting between two data representations) to non-transformational applications.

To demonstrate the applicability of our approach to real code, we developed two bidirectional libraries [**?**], one extends the attoparsec monadic parser combinator library to biprinters, and one which extends QuickCheck to bigenerators.

However, this is not the final word on sequentially composable BX programs. In all three applications, round-tripping properties are similarly split into weak round tripping, which is weaker than the original property but compositional, and purifiable, which is equationally friendly. An open question is whether an underlying structure can be formalized, perhaps based on an adjunction model, that captures bidirectionality even more concretely than monadic profunctors.

A Further code

Complete `Monad` *instance for biparsers* The instance is the straightforward product of the monad instances for `Parser` and `Printer`, where the two parts remain independent:

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

  (>>=) :: Biparser u v → (v → Biparser u w) → Biparser u w
  pu >>= kw = Biparser parse' print' where
    parse'  s = let (v, s') = parse pu s in parse (kw v) s'
    print'  u = let (v, s) = print pu u
                    (w, s') = print (kw v) u in (w, s ++ s')
```

B Proofs for compositional reasoning

The supplementary Coq proofs formalise many results of Section 4. We include some results here as hand-proofs for human consumption.

Proposition 1 The biparser `char :: Bp Char Char` (§3.2) is both backward and forward round tripping. Proof by expanding definitions and algebraic reasoning.

Proof. By straightforward expansion of the definitions. For backward round tripping:

```
fmap snd (print p x) = Just [c]
```

Then `parse p ([c] + +s') = (c, s')` (QED).

For forward round tripping, `parse p s = (x, "")` means that s must be `[x]`, then: `fmap snd (print p x) = Just [x]` (QED).

Proposition 12. The `return` operation for the `Biparser` monadic profunctor is not backward round tripping, but it is *weak* backward round tripping.

Proof. Let `x, y :: u` and `s, s' :: String`:

– (*not* backwards round tripping)

```
    fmap snd (print (return y) x)
≡ fmap snd ((λ_ → Just (y, "")) x)
≡ fmap snd (Just (y, ""))
≡ Just ""
```

Thus `s == ""`. Now we must prove the consequent of backwards round tripping, but it turns out to be false:

```
    parse (return y) ("" ++ s')
≡ (λs → (y, s)) s'
≡ (y, s')
≢ (x, s')
```

Thus, `return` is not backwards round tripping.

– (weak backwards tripping)

```
    print (return y) x
≡ (λ_ → Just (y, "")) x
≡ Just (y, "")
```

Thus `s == ""`. Now we must prove the consequent of weak backwards round tripping:

```
    parse (return y) ("" ++ s')
≡ (λs → (y, s)) s'
≡ (y, s')    □
```

Proposition 2 Weak backward round tripping of biparsers is compositional.

Proof. Case `return`. Shown above.

Case `(>>=)`.

```
let (sp, v) = print p u
    (sk, w) = print (k v) u
print (p >>= k) u = (sp ++ sk, w)          -- by definition
parse p (sp ++ sk ++ s') = (v, sk ++ s')   -- by weak round tripping of p
parse (k v) sk = (w, s')                   -- by weak round tripping of k
```

Case `comap`: trivial.

Theorem 1 If `p :: P u u` is weak backward round tripping, and for all `x :: u`. `purify p x = Just x`, then `p` is backward round tripping.

Proof. The definition of `purify p x = fmap fst (print p x)` when combined with the property `purify p x = Just x`, and the antecedent of backward round tripping (`fmap snd (print p x) = Just s`), imply that `print p x = Just (x, s)`. This satisfies the antecedent of weak backward round tripping, thus we can conclude `parse p (s ++ s') = (x, s')`, and thus backward round tripping holds for `p`.

In Section 4.1 in the proof of Corollary 3, we used three intermediate results about `char`, `int` and `replicateP`, namely:

```
proj char n ≡ Just n                                              (4.1)
proj int n ≡ Just n                                               (4.2)
proj (replicateBiparser (length xs) p) xs ≡ mapM (proj p) xs      (4.3)
```

The first two are straightforward from their definitions. Let us take a closer look at the latter, (4.3).

As a printer, `replicateBiparser (length xs)` applies the printer `p` to every element of the input list `xs`, and if we ignore the output string with `proj`, that yields `mapM (proj p) xs`. When `p` is aligned and has `proj p = Just`, as was the case in the proof of Corollary 3 then all applications in the list succeed and return a `Just` value, so `mapM (proj p) xs` as a whole succeeds and returns the whole list of results. Therefore, `replicateBiparser (length xs) p xs = Just xs`.

C Lenses

Theorem 6 If a lens `l` is weak forward round tripping and has identity projections on some subset `P` for all `s` (`P x` $\Rightarrow$ `proj s l x = Just x`) then it is also forward round tripping on `P`.

Proof. Assume the antecedent of backward roundtripping:

```
put l x s = Just ((y, s'), p') ∧ p' s' = True
```

The goal is to prove `get l s' = Just x`.

By the identity projection premise we have that `proj s l x = Just x` for all `s`. Recall the definition of `proj` for lenses:

```
proj s l = λu → fmap (fst ∘ fst) (put l u s)
```

Combining this with assumption on `put` and identity project we see that:

```
put l x s = Just ((x, s'), p')
```

We can thus instantiate weak backward round tripping to get the desired goal:

```
get l s' = Just x
```

Proposition 8 Atomic lens `atKey k` is weak backward round tripping.

Proof. Recall `atKey :: L Src Value Value`. Assuming the antecedent of backward round-tripping, we get the following information:

```
put l x (atKey k) m = Just ((x, insert k x m), λm' → lookup k m' == Just x)
∧ (λm' → lookup k m' == Just x) z'' = True
```

We then need to prove `get l z'' = Just x`. By the definition of `get`:

```
get (atKey k) z'' = lookup k z''
```

By the second conjunct of the antecedent we know `lookup k z'' = Just x`, giving the required consequent.

Proposition 9 Atomic lens `atKey k` has identity projection: `proj (atKey k) = Just`.

Proof. For all `s :: s`, following the definition we get:

```
    proj s (atKey k)
≡ λu → fmap (fst ∘ fst) (put (atKey k) u s)
≡ λu → fmap (fst ∘ fst) (Just ((u, ...), ...))
≡ λu → Just u
```

C.1 Further example: Lenses Over Trees

Our lens structuring provides the following two smart deconstructors and one smart constructor:

```
get :: L s u v → (s → Maybe v)
put :: L s u v → (u → s → Maybe ((v, s), s → Bool))
mkLens :: (s → Maybe v) → (u → s → Maybe ((v, s), s → Bool)) → L s u v
```

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

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

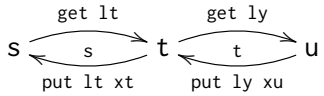
In this example, our aim is to build a lens whose forward direction gets the right spine of the tree as a list of integers. The backwards direction will then allow a tree to be updated with a new right spine (represent as a list of integers), which may produce a larger source tree.

We start by defining the classical lens combinator. Given a lens `lt` to view `s` as `t`, and a lens `ly` to view `t` as `u`, the combinator `(>>>)` creates a lens to view `s` as `u`. We illustrate and explain the composition on the right.

```
(>>>) :: L s t t → L t u u → L s u u
lt >>> ly = mkLens get' put' where
  -- get' :: s → Maybe u
  get' s = get lt s >>= get ly

  -- put' :: u → s →
  --         Maybe ((u, s), s → Bool)
  put' xu s =
   case get lt s of
     Nothing → Nothing
     Just t → do
        ((y, xt), q') ← put ly xu t
        ((_, s'), p') ← put lt xt s
        if q' xt
          then Just ((y, s'), p')
          else Nothing
```

Illustration of the composition of lenses in `(>>>)`:



For individual lenses, the `put` action takes the source as its last parameter (shown above the lower arrows here). In the case of the composite lens, `put'` has a source of type `s`, thus we need to create an intermediate source of type `t` in order to use `put ly`. This intermediate source is provided by first using `get lt s`.

In the last three lines of `putter` in `(>>>)`, in order for the composite backwards direction to succeed, the returned intermediate store `xt` must be consistent (free of conflict) as checked by `q' xt`.

We define two primitive lenses: `rootL` and `rightL` for the root and right child of a tree:

```
rootL :: L Tree (Maybe Int) (Maybe Int)
rootL = mkLens getter putter
  where
   getter (Node _ n _) = Just (Just n)
   getter Leaf         = Just Nothing

   putter n' t = Just ((n', t'), p)
    where
      t' = case (t, n') of
        (_,    Nothing) → Leaf
        (Leaf, Just n)  → Node Leaf n Leaf
        (Node l _ r, Just n)  → Node l n r
      p t'' = getter t' == getter t''
```

```
rightL :: L Tree Tree Tree
rightL = mkLens getter putter
  where
   getter (Node _ _ r) = Just r
   getter  _           = Nothing

   putter r Leaf = Nothing
   putter r (Node l n _) = Just
     ((r, Node l n r),
       λt' → Just r == getter t')
```

The `rootL` lens accesses the label at the root if it is a `Node`, otherwise returning `Nothing`. Note that `Maybe` type used here is different to use the of `Maybe` inside the definition of `L`: internally `L` uses `Maybe` to represent failure, here at the top-level we are using it to merely indicate presence of absence of a label.

The second lens `rightL` accesses the right child of a tree which can however fail if the source tree is a `Leaf` rather than a `Node`.

Both lenses provide `put` operations which return predicates that check that the view of a store is equal to the view of the store updated by the put.

We compose these two primitive lenses monadically to define the `spineL` lens to view and update the right spine of a tree:

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

Auxiliary functions `safeHead` and `safeTail` are defined:

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

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

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

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

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

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

D Generators

This appendix section provides additional code examples for our notion of *bi-generators* (*bidirectional generators*) extending random generators in property-based testing frameworks like *QuickCheck* [**?**] to a bidirectional setting.

We assume given a `Gen` monad of random generators (e.g. as defined in the QuickCheck library for Haskell) and two primitive generators: `genBool :: Double → Gen Bool` generates a random boolean according to a Bernoulli distribution with a given parameter $p \in [0, 1]$; `choose :: (Int, Int) → Gen Int` generates a random integer uniformly in a given inclusive range $[\min, \max]$.

Generators for binary search trees We consider again the type of trees from the previous section. A *binary search tree* (BST) is a `Tree` whose nodes are in sorted order. Inductively, a BST is either a `Leaf`, or some `Node l n r` where `l` and `r` are both binary search trees, nodes in `l` have smaller values than `n`, and nodes in `r` have greater values than `n`.

As a working example, we are given some function `insert :: Tree → Int → Tree` which inserts an integer in a BST. We want to test the invariant that BSTs are mapped to BSTs, by *generating* a BST and an integer to apply the `insert` function, and *check* that the output is also a BST.

With the `Gen` monad, we can write a simple generator of BSTs recursively: given some bounds on the values of the nodes, if the bounds describe a nonempty interval, we flip a coin to decide whether to generate a leaf or a node, and if it is a node, we recursively generate binary search trees, following the inductive defini-

tion above. We can similarly write a *checker* for binary search trees as a predicate.

```
genBST :: Int → Int → Gen Tree
genBST min max | max < min = return Leaf
genBST min max = do
  isLeaf' ← genBool 0.5
  if isLeaf' then return Leaf
  else do n ← choose (min, max)
          l ← genBST min (n-1)
          r ← genBST (n+1) max
          return (Node l n r)
```

```
checkBST :: Int → Int → Tree → Bool
checkBST min max Leaf = True
checkBST min max (Node l n r) =
     min ≤ n
  && n ≤ max
  && checkBST l
  && checkBST r
```

Bigenerator 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, provided here by a smart constructor: `mkAlignedG`:

```
mkAlignedG :: Gen v → (v → Bool) → G v v
mkAlignedG gen check = mkG gen (λy → if check y then Just y else Nothing)
```

Recall from Section 6 that a bigenerator can be mapped to a predicate via `toPredicate`:

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

We wrap two generator primitives as `bool` and `inRange`. As predicates, `bool` makes no assertion, `inRange` checks that the input integer is within the given range.

```
bool :: Double → G Bool Bool
bool p = mkAlignedG
  (genBool p)
  (λ_ → True)
```

```
inRange :: (Int, Int) → G Int Int
inRange (min, max) = mkAlignedG
  (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 → Maybe Int
nodeValue (Node _ n _) = Just n
nodeValue _            = Nothing

nodeLeft, nodeRight :: Tree → Maybe Tree
nodeLeft (Node l _ _) = Just l
nodeLeft _            = Nothing

nodeRight (Node _ _ r) = Just r
nodeRight _            = Nothing

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

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

We then define a specification of binary search trees (`bst` below). 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 (Just ∘ isLeaf) (bool 0.5)
  if isLeaf' then return Leaf
  else 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))
```

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.