The Monadic Profunctor Paradigm of Bidirectional Programming

LI-YAO XIA, Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, Laboratoire Méthodes Formelles, France
SAMANTHA FROHLICH, University of Bristol, UK
DOMINIC ORCHARD, University of Cambridge, UK and University of Kent, UK
MENG WANG, University of Bristol, UK

Converting data from one representation to another and vice versa is a common task in software. However, naïvely specifying both conversion directions separately is error prone and introduces conceptual duplication. *Bidirectional programming* advocates for the writing of one piece of code that can be interpreted in both directions (e.g., for a conversion and its inverse). Unfortunately, current bidirectional programming techniques generally require unfamiliar programming idioms such as restricted and specialised combinator libraries. Instead, we introduce a framework for composing bidirectional programs monadically, enabling bidirectional programming with a familiar functional programming abstraction. This abstraction leverages compositionality and equational reasoning for the verification of the required *round tripping properties* for such monadic bidirectional programs. We introduce partial monadic profunctors as a new, general paradigm of bidirectional programming, and demonstrate their use in a number of domains: parsers/printers, pickling, lenses, generators/predicates, and logic programming.

CCS Concepts: • **Software and its engineering → Functional languages**; **Data types and structures**; **Patterns**; **Frameworks**.

Additional Key Words and Phrases: bidirectional programming, profunctors, monads, compositionality, equational reasoning

ACM Reference Format:
Li-yao Xia, Samantha Frohlich, Dominic Orchard, and Meng Wang. 2025. The Monadic Profunctor Paradigm of Bidirectional Programming. *ACM Trans. Program. Lang. Syst.* 1, 1 (March 2025), 55 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A *bidirectional transformation* (BX) is a pair of mutually related mappings between source and target data objects. A classic application of this ideas is the *view-update problem* [Bancilhon and Spyratos 1981] from relational database design, where 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 can be neatly captured as a bidirectional transformation. The bidirectional pattern is found in a broad range of applications, including parsing [Matsuda and Wang 2013; Rendel and Ostermann 2010], pickling [Kennedy 2004], refactoring [Schuster et al. 2016], code

Authors' Contact Information: Li-yao Xia, li-yao.a.xia@inria.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France; Samantha Frohlich, samantha.frohlich@bristol.ac.uk, University of Bristol, Bristol, Avon, UK; Dominic Orchard, dominic.orchard@cl.cam.ac.uk, University of Cambridge, Cambridge, UK and University of Kent, Canterbury, UK; Meng Wang, meng.wang@bristol.ac.uk, University of Bristol, Bristol, Avon, UK.

ACM 1558-4593/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

generation [Mayer et al. 2018; Pombrio and Krishnamurthi 2014], model transformation [Stevens 2008], and XML transformation [Pacheco et al. 2014b].

When programming a bidirectional transformation, one can separately construct the "forward" (view) and "backward" (update) functions. However, this approach duplicates effort, is prone to error, and causes subsequent maintenance issues as both directions must be kept in sync. These problems can be avoided by using specialised programming languages that generate both directions from a single definition [Foster et al. 2007; Matsuda et al. 2007; Voigtländer 2009], a discipline known as *bidirectional programming*.

The most well-known language family for BX programming is *lenses* [Foster et al. 2007]. A lens captures transformations between sources `S` and views `V` via a pair of functions `get : S → V` and `put : V → S → 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 for `put` 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`.

Typically, bidirectional transformations 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:

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

Lens languages are typically designed to enforce these properties *by construction*. This focus on unconditional correctness inevitably leads to reduced practicality: lens combinators—the main way to program lenses—are often stylised and disconnected from established programming idioms.

In this paper, we instead focus on expressing bidirectional programs directly in idiomatic functional code, using monads as an interface for sequential composition. Monads are a popular pattern [Wadler 1995] (especially in Haskell) that combinator libraries in other (non-bidirectional) domains routinely exploit. Introducing monadic composition to BX programming significantly expands the expressiveness of BX languages and 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 [Matsuda and Wang 2013]) 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). We show, amongst many other examples, that bidirectional parsing-and-printing can be given a monadic treatment.

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 instead on monadic programming, naturally reusing host language constructs. We prove the validity of our approach by revisiting lenses and other known bidirectional domains, and test its boundaries by using it to explore completely new domains. By being uncompromising about the monadic interface, we expose the essential ideas behind our framework whilst maximising its utility. The trade off with our approach is that we can no longer enforce correctness in the same way as conventional lens languages: 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 4). Our approach is based on a construction that specifies a *Partial Monadic Profunctor (PMP)*, parameterised by two application-specific monads, which are used to generate the *forward* and

backward directions. We later generalise our approach to include a *right monadic coaction* over PMPs which captures a common related programming pattern, increasing the power of the backwards direction (Section 10).
- To demonstrate the flexibility of our approach, we apply the above method to five different problem domains: parsers/printers (Section 4 and 5), picklers (Section 6), lenses (Section 7), generators/predicates for structured data (Section 8), and logic programming (Sections 9 and 10). While parsers/printers and lenses are well-explored areas in the bidirectional programming literature, generators are a completely new application domain, which we fully explore through their generalisation that enables logic programming (Section 9).
- We present a scalable reasoning framework, capturing notions of *compositionality* for bidirectional properties (Section 5). We define classes of round tripping properties inherent to bidirectionalism, which can be verified by following simple criteria. These principles are demonstrated with our examples. We include some proofs for illustration in the paper. The supplementary material [Xia et al. 2025] contains machine-checked Rocq proofs for the main theorems.
- We have implemented these ideas as Haskell libraries [Xia 2018], 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. All of the proofs for this paper, apart from more complex examples in Section 8 and Section 9, are mechanised in Rocq [Xia et al. 2025].

Change log. This work extends "Composing bidirectional programs monadically" [Xia et al. 2019] with the following updates and additions:

- Inlining of relevant appendix material, more examples, and expanded explanations.
- Unification with the `profunctors` package[1] enabling the results of this work to be more easily applied within the Haskell ecosystem.
- New application domains: picklers [Kennedy 2004] (Section 6) and logic programming (Sections 9 and 10), where the latter is a substantial addition, generalising monadic profunctor generators to solve classic logic puzzles such as n-queens or the river crossing puzzle.
- For bigenerators, our framework previously only enabled proofs of *completeness*. To handle *soundness*, we propose *idiomatic compositionality*, a new notion of compositionality more restrictive than quasicompositionality.
- Insight into, and generalisation of, partial profunctors as *right monad coactions*, enabling the above application to logic programming (Section 10).
- [Xia et al. 2025] contains updated Haskell code and Rocq proofs.

2 Background

We introduced lenses briefly above. We now introduce *parsers/printers* and *generators/predicates*. As *picklers* build upon *parsers/printers* and our *logic programming* approach is achieved by generalising generators, these will be introduced later.

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. An (overly) simplistic implementation of these two functions can be given with types:

[1] https://hackage.haskell.org/package/profunctors

```
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 satisfying 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
```

i.e., after parsing `s` into an AST fragment `x` and remaining text `s'`, printing `x` back to a string, concatenating it with the remained `s'`, and re-parsing is just the same as the original parsing. This pattern resembles lenses between a source (source code) and view (ASTs), but with a notion of composition for the source, and partial "gets" which consume some of the source leaving a remainder. Thus, parsing and printing follows a pattern of inverse-like functions which does not fit the lens paradigm.

Writing parsers and printers by hand is often tedious due to the redundancy implied by their inverse-like relation. Thus, various approaches have been proposed for reducing the effort of writing parsers/printers by generating both from a common definition [Matsuda and Wang 2013, 2018; Rendel and Ostermann 2010]. Bidirectional parsers/printers will be the running example in the next section.

Generating and checking. Property-based testing (*e.g.* QuickCheck) [Claessen and Hughes 2000] 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 ⟹ is implication for properties:

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

To test this property, a testing framework generates random inputs for `val` and `list`. The implementation of ⟹ applied here first checks whether `list` is sorted, and if it is, checks that `insert val list` is sorted as well. This process is repeated with further random inputs 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 tested property 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 eventually generated. In this setting of testing, completeness is arguably more important than soundness, as completeness affects the potential adequacy of testing whilst soundness 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, the

ability to extract both from a common specification with bidirectional programming is a very attractive alternative. We return to this example in detail in Section 8

3 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 Haskell pun of naming variables after their types, *e.g.*, a variable of some polymorphic type `v` will also be called `v`. Similarly, for some type constructor `m`, a variable of type `m v` will be called `mv`. 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 → Maybe (v, String)}
```

Here a `Parser` consists of a function that takes a `String` (the input to be parsed), and from it *maybe* produces the resulting parsed value of type `v` paired with a `String`, which contains the remainder of the input that was not used in parsing. It is well-known that such parsers are monadic [Wadler 1995], *i.e.*, they have a notion of sequential composition embodied by the interface:

```
instance Monad Parser where
  (≫=) :: Parser v → (v → Parser u) → Parser u
  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. Indeed, this is the implementation given to the monadic interface:

```
pv ≫= ku = Parser (λs → parse pv s ≫= λ(v, s') → parse (ku v) s')
return v = Parser (λs → Just (v, s))
```

Bind first runs the parser `pv` on an input string `s`, resulting in maybe a value `v` and that rest of the string `s'`. Then, `v` is passed on using the bind of `Maybe` to create the parser `ku v`, which is in turn run on the remaining input `s'` to produce parsed values of type `u`. The `return` operation creates a trivial parser for any value `v` which does not consume any input but simply produces `v`.

In practice, parsers composed with (≫=) often have a relationship between the output types of the two operands: one "contains" the other in some sense. 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 `parser :: Parser v → (String → Maybe 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 → Maybe (String → v)
printer :: Biparser v →       (v → String)
```

However, this type of printer `v → String` (shown also in Section 2) 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 → (u → String)) → (u → String)
bind pv ku = λu → let v = (??) in pv v ++ ku v u
```

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 `ku` (second printer which depends on a `v`). Subsequently, we cannot construct a monadic biparser by simply taking a product of the parser monad and `v → String` and leveraging the result that the product of two monads is a monad.

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

```
bind' :: (u → v) → (v → String) → (v → (u → String)) → (u → String)
bind' from pv ku = λu → let v = from u in pv v ++ ku v u
```

This is closer to the monadic form, where `from :: u → 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 u`". In the context of constructing a bidirectional parser, having such a function to hand is not an unrealistic expectation: when we compose two parsers with (≫=), typically the values of the first parser for `v` will be contained within the values returned by the second parser for `u`, 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 the contextualizing projection together, `(u → v, v → String)` and fusing them into `(u → (v, String))`. This has the advantage of removing the contravariant occurence of `v`, yielding a data type:

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

If we fix the first parameter type `u`, then the type `Printer u` of printers for `u` values is indeed a monad, combining a *reader monad* (for some global read-only parameter of type `u`) and a *writer monad* (for strings) [Jones 1995], with implementation:

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

(≫=) :: Printer u v → (v → Printer u t) → Printer u t
pv ≫= kt = Printer (λu → let (v, s)  = print pv u
                             (t, s') = print (kt v) u
                         in (t, s ++ s'))
```

The printer `return v` ignores its input and prints nothing. For bind, an input `u` 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_Printer :: (u → u') → Printer u' v → Printer u v
comap_Printer from (Printer p) = Printer (p ∘ from)
```

3.1 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, their product is a monad, defining *biparsers*:

```
data Biparser u v = Biparser {biParse :: String → Maybe (v, String)
                             , biPrint :: u      →       (v, String)}
```

By pairing parsers and printers we have to unify their covariant parameters. When both the type parameters of `Biparser` 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:

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

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

In the above instance, the product construction is in evidence: in both `return` and `(≫=)`, the `Biparser` constructor is provided with first the `Parser` version of the function and then the `Printer` version.

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

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

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

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 the function `subpart :: u → u'`, thus yielding a biparser of type `Biparser u v`.

Often, the desired `subpart` function, that "inverts" the forward computation, is partial. For example, when `u` is a sum type (and the subpart in question is only one side of the sum), or when `u` is a list that needs decomposing (e.g., by `head`). Thus the more appropriate type is `subpart :: u → Maybe u'`. We thus adjust the type of $\texttt{comap}_{\texttt{Biparser}}$ accordingly. To avoid handling `Maybe` within each `Biparser`, we also add the possibility of failure to the backward direction:

```
data Biparser_M u v = Biparser_M {biParse_M :: String → Maybe (v, String)
                                 , biPrint_M :: u      → Maybe (v, String)}

comap_BiparserM :: (u → Maybe u') → Biparser_M u' v → Biparser_M u v
comap_BiparserM f (Biparser_M parse print) = Biparser_M parse (λu → f u >>= print)

upon_M :: Biparser_M u' v → (u → Maybe u') → Biparser_M u v
upon_M = flip comap_BiparserM
```

An example biparser. Let us write a biparser, `string :: Biparser_M String String`, for strings that are prefixed by their length and a space. For example, the following unit tests should succeed:

```
test1 = biParse_M string "6 lambda calculus" ≡ Just ("lambda", " calculus")
test2 = biPrint_M string "SKI" ≡ Just ("SKI", "3 SKI")
```

In `test1`, the parser consumes the number 6 followed by a space. This tells it to gobble the next six characters following the space. Hence, the expected result is the six letters of `"lambda"` to be parsed, and the rest to be left. When it comes to the printer, `test2` presents the word `"SKI"`. Since this is three letters long, the printer will print the number three and a space followed by the word. For brevity, we will not handle failing cases, assuming inputs always have the expected form.

We start by defining a primitive biparser of single characters as:

```
char :: Biparser_M Char Char
char = Biparser_M parseChar (λc → Just (c, [c]))
  where parseChar []    = Nothing
        parseChar (c:s) = Just (c, s)
```

A character is parsed by deconstructing the source string into its head and tail. A character `c` is printed as its single-letter string (a singleton list) paired with `c`.

In our example, we first need to parse the number. We will do this by parsing the digits of the number using `char` until there is a space, using the combinator `manyUntilBP`:

```
manyUntilBP :: Biparser_M u a → (a → Bool) → Biparser_M [u] [a]
manyUntilBP one p = do
  x ← one `upon_M` head_M
  if p x
    then return [x]
    else do
      xs ← (manyUntilBP one p) `upon_M` tail_M
      return (x:xs)
```

where $\texttt{head}_M$ :: `[a] → Maybe a` and $\texttt{tail}_M$ :: `[a] → Maybe [a]`.

This biparser takes as input a biparser to repeat, and a condition on what it is parsing to signal when to stop the repetition; it uses the given parser to parse one example, then if the condition is met, it stops, otherwise it continues. Note that in Haskell, the **do**-notation statement "$\texttt{x} \leftarrow \texttt{one}\ `\texttt{upon}_M`\ \texttt{head}_M$" desugars to "$\texttt{one}\ `\texttt{upon}_M`\ \texttt{head}_M \gg\!= \lambda\texttt{c} \rightarrow \ldots$", which uses (>>=) and a function binding `c` in the scope of the rest of the desugared block. In the case of parsing digits to make up an integer, we want to keep parsing until we reach a space:

```
int :: Biparser_M Int Int
int = do
```

```
  ds ← manyUntilBP char isSpace `upon_M` printedInt
  return (read ds)
  where
    printedInt n = Just $ show n ++ " "
```

After parsing an integer `n`, we can parse the string following it by iterating the biparser `char` `n` times. This is captured by the `replicateBP` combinator below. This combinator is akin to `replicateM` from Haskell's standard library. To iterate `n` times a biparser `pv`: if $n \equiv 0$, 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, as a printer, `replicateBP n pv` expects a list, `l`, of length `n`: if $n \equiv 0$, 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.

```
replicateBP :: Int → Biparser_M u v → Biparser_M [u] [v]
replicateBP 0 pv = return []
replicateBP n pv = do
  v  ← pv `upon_M` head_M
  vs ← (replicateBP (n − 1) pv) `upon_M` tail_M
  return (v:vs)
```

We can now fulfil our example task to parse and print length-prefixed strings:

```
string :: Biparser_M String String
string = int `upon_M` (Just ∘ length) ⋙= λn → replicateBP 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 annotated using `upon` to access the relevant subparts of the input when running the *backward* direction of printing.

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 that is 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.

4 A unifying structure: partial monadic profunctors

The biparser examples of the previous section were enabled by both the monadic structure of `Biparser` and its covariant functor structure provided by the $\texttt{comap}_{\texttt{BiparserM}}$ operation (also called $\texttt{upon}_{\texttt{M}}$, with flipped arguments). We call the combined structure a *Partial Monadic Profunctor (PMP)*, which we view as the conjunction of two algebraic structures: monads, of course, and profunctors [Kmett 2011; Pickering et al. 2017], which have partiality injected into the first type parameter. Here, we will show that PMPs characterise a key class of structures for bidirectional programs. Furthermore, we show a construction of PMPs from pairs of monads which elicits the necessary structure for monadic bidirectional programming in the style of the previous section.

Definition 4.1. A binary type constructor `p` is a *profunctor* if it is contravariant in its first parameter and covariant in its second. Its operation given by the following class:

```
class Profunctor p where
  dimap :: (a → b) → (c → d) → p b c → p a d
```

Profunctors are subject to the laws of a bifunctor (contravariant functoriality in the first argument and covariant functoriality in the second):

```
dimap id id = id          dimap (g ∘ f) (h ∘ k) = dimap f h ∘ dimap g k
```

In category theory terminology, a profunctor is more generally a functor $\mathcal{D}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$, from the product of two categories $\mathcal{D}^{\mathrm{op}}$ and $\mathcal{C}$ to the category of sets. The naming in Haskell, coming from the existing `Profunctor` class [Kmett 2011], derives from this terminology, but instead essentially describes profunctors $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ if we identify Haskell types in an idealised way with sets. We adopt this programming-oriented terminology, and utilise Haskell's `profunctor` package.

Every profunctor is a functor in its second parameter. The following function, `rmap`, can be used to an implement `fmap` in the `Functor` class. Provided the functor laws are satisfied, `rmap = fmap` is guaranteed by parametricity:

```
rmap :: Profunctor p ⇒ (a → b) → p u a → p u b
rmap = dimap id
```

We introduce the following subclass of `Profunctor` to enable partiality.

Definition 4.2. A *partial profunctor* `p` is a profunctor equipped with a natural transformation `internaliseMaybe`:

```
class Profunctor p ⇒ PartialProfunctor p where
  internaliseMaybe :: p u v → p (Maybe u) v
  comap :: PartialProfunctor p ⇒ (u → Maybe u') → p u' v → p u v
  comap f = dimap f id ∘ internaliseMaybe
```

`internaliseMaybe` subject to the following laws, where `fmap`, `return` and `join` are specialised to the `Maybe` monad:

```
internaliseMaybe ∘ dimap f g = dimap (fmap f) g ∘ internaliseMaybe
dimap return id ∘ internaliseMaybe = id
dimap join   id ∘ internaliseMaybe = internaliseMaybe ∘ internaliseMaybe
```

The first law is the naturality of `internaliseMaybe`. That law is satisfied "for free" by parametricity [Wadler 1989]. The last two laws regulate the interaction of the profunctor `p` with the `Maybe` monad. Using `internaliseMaybe` and `dimap`, we can define the `comap_M` operation of the previous section to map over the first type parameter of `p` with a Kleisli arrow in the `Maybe` monad. This is a default method of the class.

As a consequence of the partial profunctor laws, `comap` obeys the following functor laws:

```
comap return = id
comap (f <=< g) = comap f ∘ comap g
```

Definition 4.3. A *Partial Monadic Profunctor (PMP)* is a partial profunctor `p` (in the sense of Definition 4.2) such that `p u` is a monad for all `u`, and `dimap f id` is a monad morphism for all `f :: u' → u`:

```
dimap f id (u >>= k) = (dimap f id u) >>= ((dimap f id) ∘ k)
dimap f id (return x) = return x
```

Those equations follow from parametricity, so we may represent PMPs with the following empty class, or "class synonym", which inherits all its methods from its superclasses, and which is instantiated whenever its superclasses are. The constraint `∀u.Monad (p u)` is enabled by the `QuantifiedConstraints` extension [Bottu et al. 2017].

```
class (PartialProfunctor p, ∀u.Monad (p u)) ⇒ Profmonad p
instance (PartialProfunctor p, ∀u.Monad (p u)) ⇒ Profmonad p
```

One might also expect `comap` to be a monad morphism. However, this is not the case for the `Printer` PMP of the previous section—nor for `Bwd` which will generalise `Printer`, in Section 4.1.

Note that `dimap` is uniquely determined by `comap` and `fmap`, and `internaliseMaybe` is uniquely determined by `comap`:

```
dimap f g = comap (Just ∘ f) ∘ fmap g
internaliseMaybe = comap (λ_ → Nothing)
```

Therefore, an implementation of a PMP can be derived from `return`, `(≫=)`, and `comap`.

Biparsers have lawful instances of `Monad (Biparser u)` as defined in Section 3.1 (p. 7) and an instance `Profunctor Biparser` thus defined:

```
instance Profunctor Biparser where
  dimap f g (Biparser parse print) = Biparser parse' print'
    where
      print' = λu → let (v, s) = print (f u) in (g v, s)
      parse' = λs → let (v, s) = parse s in (g v, s)
```

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

4.1 Constructing PMPs

Our examples (parsers/printers, picklers, lenses, generators/predicates, and generalise generators) all share PMPs as an abstraction, making it possible to write different kinds of bidirectional transformations monadically. Underlying these definitions of PMPs is a common structure, which we explain here using biparsers, and which will be replayed in subsequent sections for each of our examples, starting with picklers in Section 6, which is very similar to the parsers/printers construction, and ending with bigenerators and their generalisation in Sections 8 and 9.

There are two simple ways in which a functor, `m` (resp. a monad) gives rise to a profunctor (resp. a PMP). The first is by constructing a profunctor in which the contravariant parameter is discarded, *i.e.*, `p u v = m v`; the second is as a 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 ⇒ Profunctor (Fwd m) where
  dimap _ g (Fwd mx) = Fwd (fmap g mx)

instance Functor m ⇒ Profunctor (Bwd m) where
  dimap f g (Bwd kx) = Bwd ((fmap g) ∘ kx ∘ f)
```

Since the `Fwd` construction ignores the contravariant type parameter, there is nothing to apply `dimap`'s first argument to, meaning that the `Fwd` type can be passed through unchanged.

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

```
instance Monad m
   ⇒ Monad (Fwd m u) where
  return x = Fwd (return x)
  Fwd py ≫= kz =
    Fwd (py ≫= unFwd ∘ kz)
```

```
instance Monad m
   ⇒ Monad (Bwd m u) where
  return x = Bwd (λ_ → return x)
  Bwd my ≫= kz = Bwd
    (λu → my u ≫= (λy → unBwd (kz y) u))
```

The `Fwd` is trivially a partial profunctor:

```
instance Functor m ⇒ PartialProfunctor (Fwd m) where
  internaliseMaybe (Fwd x) = Fwd x
```

For `Bwd` to be a partial profunctor, we require that the monad `m` has a notion of failure, via the following `MonadFail` class (which is part of the Haskell standard library):

```
class Monad m ⇒ MonadFail m where fail :: String → m a
```

Instances of the type class are expected to obey the following axiom for all `s :: String` (which represents an error message):

```
(fail s ≫= k) = fail s
```

If `m` is a monad with a `fail`, we can define `internaliseMaybe` by producing `fail` when the input is `Nothing`.

```
instance MonadFail m ⇒ PartialProfunctor (Bwd m) where
  internaliseMaybe (Bwd k) = Bwd k' where
    k' Nothing = fail "Error"
    k' (Just x) = k x
```

Standard instances. The `Maybe` type constructor is the prototypical instance of `MonadFail`:

```
instance MonadFail Maybe where fail _ = Nothing
```

For every monad transformer `t`, we also get an instance `MonadFail m ⇒ MonadFail (t m)`. We define it as an individual instance for each transformer to avoid overlapping instances.

```
instance (Monoid w, MonadFail m) ⇒ MonadFail (WriterT w m) where
  fail e = WriterT (fail e)

instance MonadFail m ⇒ MonadFail (StateT s m) where
  fail e = StateT (λ_ → fail e)
```

The product of two PMPs is also a PMP. This follows from the fact that the product of two monads is a monad and the product of two functors is a 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 (Profunctor p, Profunctor q) ⇒ Profunctor (p:*:q) where
  dimap f g (py:*:qy) = dimap f g py:*:dimap f g qy
```

These instances are a simple dance around the `(:*:)` constructor that sticks the two monads together, applying the corresponding underlying instance to the correct monad.

Finally, the product of two partial profunctors is a partial profunctor:

```
instance (PartialProfunctor p, PartialProfunctor q)
   ⇒ PartialProfunctor (p:*:q) where
  internaliseMaybe (py:*:qy) = (internaliseMaybe py):*:(internaliseMaybe qy)
```

Therefore, we can form PMPs by a pair of `Fwd` and `Bwd` constructions.

4.2 Deriving biparsers as PMP pairs

We can redefine biparsers in terms of the above data types, their instances, and two standard monads, the state and writer monads (we are using the monad transformer versions, so the results can be wrapped in `Maybe`):

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

The forward component corresponds to parsers, and the backward to printers. Both sides may fail (`Maybe`): the parser because parsing is a partial process; the printer because it is necessary to obtain a partial contravariant map `comap` via the `PartialProfunctor` class.

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

We define smart constructors and destructors for accessing the two components of biparsers:

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

The PMP definition for biparsers now comes for free from the constructions in Section 4.1. The example biparsers from Section 3.1 can be easily redefined using the structure here. For example, the primitive biparser `char` becomes:

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

In a similar manner, we will use this PMP construction to define monadic bidirectional transformations for our other examples.

Codec library. The codec library [Chilton] 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. In particular, the second component was generalised from `codecOut :: c → w ()` in order to support a `Monad` instance for monadic bidirectional programming with codecs.

5 Reasoning about bidirectionality

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

This section studies the interaction of PMPs 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 backward-then-forward) and *forward round tripping* (going forward-then-backward). In each BX domain, such properties also capture additional 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. For brevity, we use `Bp` as an alias for `Biparser`. We separate the string and value components of printers via helper functions:

```
printS :: Bp u v → u → Maybe String
printS p = fmap snd ∘ print p

printV :: Bp u v → u → Maybe v
printV p = fmap fst ∘ print p
```

To begin, we consider the simpler case of *aligned* biparsers (Definition 4.4), of type `Bp u u`, using the same type `u` for both the input of the printer and for the output of the parser. When the inputs and outputs of both direction are thus aligned, it is straightforward to compose them in one of two ways, and equate the output with the input.

Starting with a value `x`, print it (going "backward") into a string `s`, and parse that string, the result should be equal to the initial value `x`. In the following definitions, underscores abbreviate existentially quantified values; `parse p s = (x, _)` is shorthand for `∃s', parse p s = (x, s')`.

Definition 5.1. A biparser `p :: Bp u u` is *backward round tripping* if, for all `x :: u` and `s :: String`, this implication holds:

```
print p x = Just (_, s)   ⟹   parse p s = Just (x, _).
```

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

Definition 5.2. A biparser `p :: Bp u u` is *forward round tripping* if for every `x :: u` and `s :: String`, this implication holds:

```
parse p s = Just (x, "")   ⟹   print p x = Just (_, s).
```

Note, in some applications, that definition of 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. Coarser equivalence relations on strings could enable more flexible forward round tripping properties for such cases.

The biparser `char :: Bp Char Char` is aligned, so we can check its round tripping properties.

Proposition 5.3. The biparser `char` (§4.2) is both backward and forward round tripping.

Proof. By expanding definitions and algebraic reasoning. For backward round tripping, if `print char c = Just (_, [c'])`, then `c = c'` and we can show by the expanding of definitions that `parse char [c'] = Just (c, _)`:

```
    parse char [c']
≡   (λ(c0:s0) → Just (c0, s0)) [c']          { Def. parse char, (:) case }
≡   Just (c', "")                            { β-reduction }
≡   Just (c, "")                             { Assumption }
```

For forward round tripping, `parse p s = Just (c, "")` means that `s` must be `[c]`:

```
    print char c
≡   (λc0 → Just (c0, [c0])) c                { Def. print char }
≡   Just (c, [c])                            { β-reduction }
```

□

5.1 Compositional properties of monadic bidirectional programming

As we compose parsers from smaller pieces, one naturally expects to prove round tripping properties in a similarly modular manner. A property is *compositional* with respect to a PMP if it is preserved by its operations (`return`, (≫=), `dimap`, and `internaliseMaybe`, where `comap` is also sufficient to cover the last two). Compositional properties are guaranteed "by construction" for programs built from those components.

Definition 5.4. A *property* $\mathcal{R}$ over a PMP `P` is a family of subsets $\mathcal{R}^{u}_{v} \subseteq$ `P u v` indexed by types `u` and `v`.

Definition 5.5. A property $\mathcal{R}$ is *compositional* with respect to the operations of a (partial) PMP if the following conditions hold for all types `u`, `v`, `w`, for all `x :: v`, `p :: P u v`, `k :: v → P u w`, `f :: u' → Maybe u`,

$$\texttt{(return x)} \in \mathcal{R}^{u}_{v} \qquad \text{(comp-return)}$$

$$\left(\texttt{p} \in \mathcal{R}^{u}_{v}\right) \wedge \left(\forall \texttt{x}.\ \texttt{(k x)} \in \mathcal{R}^{u}_{w}\right) \implies \texttt{(p >>= k)} \in \mathcal{R}^{u}_{w} \qquad \text{(comp-bind)}$$

$$\texttt{p} \in \mathcal{R}^{u}_{v} \implies \texttt{(comap f p)} \in \mathcal{R}^{u'}_{v} \qquad \text{(comp-comap)}$$

Unfortunately, forward and backward round tripping as defined above for biparsers are *not* compositional. For one, compositionality considers biparsers which might not even be aligned. Furthermore, even when starting with round tripping biparsers, arbitrary compositions using the above operations are not backward round tripping. For instance, consider a biparser `char ≫= λz → return (succ z)`. As a parser, it can parse any character and return its successor, *e.g.*, `"a"` is parsed to `'b'`, but as a printer it just prints the given character, *e.g.*, `'b'` is printed as `"b"`, different from the original `"a"`.

Counterexample 5.6. The `return` operation is not backward round tripping for `Biparser`.

Proof. For `return` to be backward round tripping the following must hold:

```
print (return y) x = Just (_, s)   ⟹   parse (return y) s = Just (x, _).
```

Let `x`, `y :: u` and `s`, `s' :: String`, then we proceed by the following reasoning:

```
    print (return y) x
≡   (λ_ → Just (y, "")) x                    { Def. return }
≡   Just (y, "")                             { β-reduction }
```

Thus `s ≡ ""`. However the consequent of backward round tripping does not hold:

```
    parse (return y) ""
≡   (λs → Just (y, s)) ""                    { Def. parse (return y) }
≡   Just (y, "")                             { β-reduction }
 ≢  Just (x, "")
```

Thus, `return` is not backward round tripping. □

This suggests a need for more general properties that capture the full gamut of possible biparsers.

We first focus on backward (printing-then-parsing) round tripping. To obtain a more compositional version, we generalise it by adding an arbitrary suffix before parsing. The parser should not touch or look at the suffix: the parser is *context agnostic*.

We also allow the two type parameters of `Bp` to vary independently. If we print (going backward) and parse the resulting output (going forward) then we should 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. Our actual intent for printers is that what we print is a fragment of `u`, which is produced as the output of the printer along with the string. By thus comparing the outputs of both the parser and printer, we obtain the following variant of backward round tripping:

Definition 5.7. 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') = Just (y, s')
```

This differs from backward round tripping (`printS p x = Just s ⟹ parse p (s ++ s') = Just (x, s')`) in that we switch to using `print` (instead of `printS`), giving us access to `y`, which we now compare to instead of the original value `x`.

Weakening the backward round tripping property in that way allows `return` to satisfy it:

Lemma 5.8. The `return` operation for the `Biparser` PMP is weak backward round tripping.

Proof. For `return` to be *weak* backward round tripping the following must hold:

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

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

```
    print (return y) x
≡   (λ_ → Just (y, "")) x                    { Def. print (return y) }
≡   Just (y, "")                             { β-reduction }
```

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

```
    parse (return y) ("" ++ s')
≡   parse (return y) s'                      { Def. (++) }
≡   (λs → Just (y, s)) s'                    { Def. parse (return y) }
≡   Just (y, s')                             { β-reduction }
```

□

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 5.9. Weak backward round tripping of biparsers is compositional.

Proof. We consider in turn the three cases that need to hold for compositionality:
Case `return`: Shown above.
Case (≫=):
Assume the weak round tripping of `p`:

```
print p u = Just (w, sp)
  ⟹   parse p (sp ++ s') = Just (w, s')
```

Assume the weak round tripping of `k` for all `v`:

```
print (k v) u = Just (w, sk)
  ⟹   parse (k v) (sk ++ s') = Just (w, s')
```

Our goal is to prove:

```
print (p >>= k) x = Just (w, s)
  ⟹   parse (p >>= k) (s ++ s') = Just (w, s')
```

Assume premise: `print (p >>= k) x = Just (w, s)`.
Before proving our goal, we work on the premise to learn about the shape of `s` and discharge the premises of our assumptions:

```
    print (p >>= k) u
≡   (w, sp ++ sk)                          { Def. (>>=) }
```

(Part of `s` comes from `p`, and part from `k`, we name these `sp` and `sk`.) We can now achieve our goal:

```
    parse (p >>= k) (sp ++ sk ++ s')
≡   parse (k v) (sk ++ s')                 { Weak round tripping of p }
≡   Just (w, s')                           { Weak round tripping of k }
```

Case `comap`:
For `f :: u → Maybe u'` and `p :: P u' v`, our goal is to prove:

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

Assume that `p` is weak backward round tripping, that is:

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

Assume the antecedent: `print (comap f p) x = Just (y, s)`.
Before proving our goal, we work on this antecedent to discharge the premises of our assumptions:

```
    print (comap f p) x = Just (y, s)
≡   f x = Just y'                          { Partial function succeeded }
≡   print p y' = Just (y, s)               { Simplify }
```

Thus `parse p (s ++ s') = Just (y, s')` holds by (∗), and we conclude:

```
    parse (comap f p) (s ++ s')
≡   parse p (s ++ s')                      { Def. }
≡   Just (y, s')                           { Def. + β }
```

(Recall that `comap` does nothing in the forward (in this case parsing) direction.) □

Proposition 5.10. The primitive biparser `char` is weak backward round tripping, and `replicateBP` preserves weak backward round tripping.

We can then compositionally show weak backward round tripping for a composite biparser:

Example 5.11. `string` is weak backward round tripping. (By Proposition 5.9 and Proposition 5.10.)

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 PMP combinators do not enforce a relationship between `u` and `v`. However, we can regain 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 :: ∀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 5.12. 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)` 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') = Just (x, s')`, so backward round tripping holds for `p`. □

Example 5.13. The biparser `string` is backward round tripping. (By Theorem 5.12 with Example 5.11 and Proposition 5.15.)

Proof. First we need to prove the following properties of biparsers `char`, `int`, and `replicateBP :: Int → Bp u v → Bp [u] [v]` (writing `proj` for `purify`):

```
proj char n ≡ Just n                                          (1)
proj int n ≡ Just n                                           (2)
proj (replicateBP (length xs) p) xs ≡ mapM (proj p) xs        (3)
```

The first two are straightforward from their definitions, here is the first, the second is similar:

```
    proj char n
≡   fmap fst (print char n)          { Def. proj }
≡   fmap fst (Just (n, [n]))         { Def. print }
≡   Just n                           { Def. fmap + β-reductions }
```

Let us take a closer look at the final equation, Equation 3. As a printer, `replicateBP (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`, 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, `replicateBP (length xs) p xs = Just xs`. From these and the homomorphism properties we can prove `proj string = Just`:

```
    proj string xs
≡   proj (comap length int ≫= λn → replicateBp n char) xs        { Defs. }
≡   (comap length (proj int) ≫= λn → proj (replicateBp n char)) xs { Prop. 5.15 }
≡   (comap length Just ≫= λn → proj (replicateBp n char)) xs     { Eq. (2) }
≡   proj (replicateBp (length xs) char) xs                       { Def. 4.3 }
≡   mapM (proj char) xs                                          { Eq. (3) }
≡   mapM Just xs                                                 { Eq. (1) }
≡   Just xs                                                      { Def. monad }
```

Combining `proj string = Just` with Example 5.11 (`string` is weak backward round tripping) enables Theorem 5.12, proving that `string` 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, `purify` is a *PMP homomorphism*: it commutes with the PMP operations.

Definition 5.14. A *PMP homomorphism* between PMPs `P` and `Q` is a polymorphic function `proj :: P u v → Q u v` such that:

```
proj (comap_P f p)  ≡  comap_Q f (proj p)
  proj (p ≫=_P k)   ≡  (proj p)  ≫=_Q  (λx → proj (k x))
 proj (return_P x)  ≡  return_Q x
```

Proposition 5.15. The `purify :: Bp u v → u → Maybe v` operation for biparsers (above) is a PMP homomorphism between `Bp` and the PMP `PartialFun u v = u → Maybe v`.

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

Definition 5.16. A *purifiable PMP* is a PMP `P` equipped with a homomorphism `proj` from `P` to the PMP of partial functions `– → Maybe –`. We say that `proj p` is the *pure projection* of `p`.

Definition 5.17. A pure projection `proj p :: u → Maybe u` is called the *identity projection* when `proj p x = Just x` for all `x :: u`. We say that `p` *projects to the identity.*

Here and for the other examples, 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.

Sometimes, a computation is only expected to work on a subset of the values of a type. For example, biparsers in Section 3 may only parse a subset of values within the type and in Section 8 a generator for binary trees `bst` may generate or accept only the subset of binary trees which are binary *search* trees. In those cases, the pure projection is only a restriction of the identity function.

Definition 5.18. Let `p :: P u u` and let `v` be a subset of `u`. We say that `p` *projects to the identity on* `v` if, for all `x ∈ v`, `proj p x = Just x`.

A related issue may arise when proving that a computation `p ≫= k` satisfies a compositional property $\mathcal{R}$. While Definition 5.5 requires `p` $\in \mathcal{R}^{u}_{v}$ and $\forall$`x. (k x)` $\in \mathcal{R}^{u}_{w}$, it might be that `p` only produces values in a subset of `v` and `k` only behaves well on that same subset. A possible solution

is to use more precise types, choosing `v` so that `p :: P u v` may in fact produce all values in `v`. Another solution is to index the property $\mathcal{R}$ by arbitrary subsets to describe *a posteriori* the set of elements actually produced by a computation.

5.2 Quasicompositionality for PMPs

We have neglected forward round tripping, which is not compositional, not even in a weakened form. However, we can generalise compositionality to enable forward round tripping. We call this generalised metaproperty *quasicompositionality*.

This generalisation of compositionality that admits a proof of forward round tripping depends on *injectivity*. An injective function $f : A \to B$ is a function for which there exists a left inverse $f^{-1} : B \to A$, so that $f^{-1} \circ f = id$, *i.e.*, every element in A has a unique image in B by f. We can see this pair of functions as a simple kind of bidirectional program, with a forward round tripping property (assuming f is the forward direction). We can lift the notion of injectivity to the PMP setting and capture forward round tripping properties that are preserved by the PMP operations, given an additional injectivity-like restriction. We first formalise the notion of an *injective arrow*. An injective arrow `k :: v → m w` produces an output from which the input can be recalculated:

Definition 5.19. Let `m` be a monad. A function `k :: v → m w` is an *injective arrow* if there exists `k' :: w → v` (the *left arrow inverse* of `k`) such that for all `x :: v`:

$$\texttt{k x} \gg\!= \lambda\texttt{y} \to \texttt{return (x, y)} \quad \equiv \quad \texttt{k x} \gg\!= \lambda\texttt{y} \to \texttt{return (k' y, y)}$$

Next, we define *quasicompositionality* which extends the compositionality meta-property with the requirement for $(\gg\!=)$ to be applied to injective arrows:

Definition 5.20. Let `P` be a PMP. 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, for all `x :: v`, `p :: P u v`, `k :: v → P u w`, `f :: u' → Maybe u`:

$$\texttt{(return x)} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}} \qquad \text{(qcomp-return)}$$

if `k` is an injective arrow,

$$\left(\texttt{p} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}\right) \wedge \left(\forall \texttt{x}.\ \texttt{(k x)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}}\right) \implies \texttt{(p} \gg\!= \texttt{k)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}} \qquad \text{(qcomp-bind)}$$

$$\texttt{p} \in \mathcal{R}^{\texttt{u'}}_{\texttt{v}} \implies \texttt{(comap f p)} \in \mathcal{R}^{\texttt{u}}_{\texttt{w}} \qquad \text{(qcomp-comap)}$$

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 outputs that same value `y` along with a string `s0`, then `s0` is the prefix of `s01` that was consumed by the parser, *i.e.*, `s01 = s0 ++ s1` for some `s1`.

Definition 5.21. A biparser `p:Bp u v` is *weak forward round tripping* if the following holds for all `x :: u`, `y :: v`, and `s0`, `s1`, `s01 :: String`:

```
parse p s01 = Just (y, s1)  ∧  print p x = Just (y, s0)  ⟹  s01 = s0 ++ s1
```

Proposition 5.22. 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 round tripping.

From `parse (m ≫= k) s01 = Just (y, s1)`, it follows that there exists `z`, `s` such that `parse m s01 = Just (z, s)` and `parse (k z) s = Just (y, s1)`. Similarly, the second conjunct `print (m ≫=`

`k) x = Just (y, s0)` implies there exists `{z', s0'` such that `print m x = Just (z', s0')` and `print (k z') x = Just (y, s1')` and `s0 = s0' ++ s1'`.

Because `k` is an injective arrow, we have `z = z'`. This comes from the following: recall we have: `parse (k z) s = Just (y, s1)` and `print (k z') x = Just (y, s1')`, let `k'` be the left arrow inverse of `k`, then (from Definition 5.19):

$$(\texttt{k z} \gg\!= \lambda \texttt{y} \rightarrow \texttt{return (z, y)}) = (\texttt{k z} \gg\!= \lambda \texttt{y} \rightarrow \texttt{return (k' y, y)})$$

(and similarly with `z'`). Plugging both sides of this equation into "`parse _ s`" and "`print _ x`" respectively, and using the fact that `print` and `parse` are monad morphisms, followed by the above two equations, yields the following equalities:

$$\texttt{parse (...) s = Just ((z, y), s1) = Just ((k' y, y), s1)}$$
$$\texttt{print (...) x = Just ((z', y), s1') = Just ((k' y, y), s1')}$$

Thus we can conclude (by injectivity of data constructors) that `z = (k' y) = z'`.

We then use the assumption that `m` and `k` are weak forward round tripping on `m` and on `k a`, and deduce that `s01 = s0' ++ s` and `s = s1' ++ s1`. Therefore `s01 = s0' ++ (s1' ++ s1)` which reassociates to `s01 = (s0' ++ s1') ++ s1` which equals `s01 = s0 ++ s1` (since `s0 = s0' ++ s1'`, shown above). □

Proposition 5.23. The `char` biparser is weak forward round tripping, and `manyMaybe` preserves weak forward round tripping.

Corollary 5.24. `string` is weak forward round tripping (by Propositions 5.22 and 5.23)

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 and check that arrows are injective. Similarly to backward round tripping, we can prove forward round tripping by combining weak forward round tripping with the identity projection property:

Theorem 5.25. 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 5.26. The biparser `string` is forward round tripping by the above theorem (with identity projection shown in the proof of Example 5.13) and Corollary 5.24.

5.3 Reasoning Recap

In summary, for any bidirectional transformation, we can consider two round tripping properties: foward-then-backward and backward-then-forward, called just *forward* and *backward* here respectively. Whilst combinator-based approaches usually guarantee round tripping by construction, we must compromise to get greater expressivity in the monadic approach. To reason about bidirectional transformations in a manageable, scalable way, it is desirable for round tripping properties to be compositional. Unfortunately, due to the PMP structuring, this tends not to be the case. Instead, weakened round tripping properties can be compositional or quasicompositional (adding injectivity). We then recover the stronger property by proving a simple property on aligned transformations: that the backward direction faithfully reproduces its input as its output (*identity projection*). The next subsection, compares this reasoning approach to a proof of backward round tripping for separately implemented parsers and printers (not using our combined monadic approach).

5.4 For comparison: separately defined parser and printer, and round tripping proof

The following provides a comparison between the PMP biparser of Section 3 and the alternative without our approach: having to write two separate definitions of a parser and a printer. With these examples, we also compare our reasoning approach of Section 5 with having to manually prove round tripping on the separated definitions. Appendix A shows the full definitions and proofs for comparison. The main points of comparison are:

- This parser has the same structure as the biparser in Section 3 (without any upon annotations).
- The standalone printer for this example is extremely simple, in line with a common pattern that most of the benefit of bidirectional programming is felt in the reduction of proof and maintenance effort.
- The auxiliary lemmas for `parseInt`, `parseDigits`, and `replicateM` correspond to the round tripping properties of their bidirectional counterparts (`int` and `replicateBP`).
- These lemmas must all manipulate source strings explicitly, whereas our framework uses compositionality to handle those details, which reduces the proof burden by modularisation.

6 Monadic bidirectional programming for picklers

The `BiParser` PMP can be tweaked to unlock a similar bidirectional domain. Exchanging `String` for `Endo String` using Cayley's theorem [Jacobson 2012] yields a monadic bidirectional representation of picklers:

```
Fwd (StateT String Maybe):*:Bwd (WriterT       String  Maybe)
Fwd (StateT String Maybe):*:Bwd (WriterT (Endo String) Maybe)
```

Picklers transform data to be stored or transmitted. They perform both the preparation of the data to be stored/sent ("serialisation" or "marshalling") and the reversing of the process at the other end. Similarly to parsers, this domain converts between structured and unstructured data (we'll use a `String` as our example), where just like for biparsers, we are working in the simplified setting where both directions succeed.

```
pickle : v → String        unpickle : String → Maybe v
```

6.1 Construction

```
type BiPi = Fwd (StateT String Maybe):*:Bwd (WriterT (Endo String) Maybe)
```

The augmented `BiParser` PMP packages up pickler forward and backward operations, which can be made easily accessible with smart destructors:

```
unpick :: BiPi u v → (String → Maybe (v, String))
pick   :: BiPi u v → (u → Maybe (v, Endo String))
```

The forward direction is identical to parsing: unstructured `String` data is maybe decoded into a the value that was pickled paired with the remainder of the input that was not used. In the backward direction, a value is pickled and returned paired with a series of `String` changes. These can be thought of as the pickled value, as that is what applying them to the empty string yields. Note that partiality has been artificially injected into the pickling operator to allow for the partial and contravariant `comap` operation of PMPs. The original pickler interface, modulo this partiality, can be recovered by applying the series of changes to the empty string, and ignoring the second argument of `unpick`:

```
pickle :: BiPi u v → u → Maybe String
pickle p value
  = let
    f :: (v, Endo String) → String
    f (v, Endo g) = g []
    in fmap f (pick p value)
```

```
unpickle :: BiPi u v → String → Maybe v
unpickle p s = fmap fst $ unpick p s
```

6.2 Pickler Combinators

Existing combinator libraries for picklers, such as "Pickler Combinators" [Kennedy 2004], have already inadvertently almost used PMPs, with operations and combinators that correspond to our framework. They actually only consider the simpler case where unpickling always succeeds, but since our framework make it easy to add partiality to the forwards direction, we have done so.

They present a pickler as two functions packaged up together using record syntax:

```
data PU a s = PU {appP :: (a, s) → s, appU :: s → (a, s)}
```

Squinting at these operations, they are each monads: `appP` can be refactored to become the writer monad over endomorphisms `(a, s → s)`; and `appU` requires no changes to be the state monad. Using these observations, we arrive at the Cayley theorem adjusted type from earlier: a bidirectional pickler in our framework.

Their combinators also featured ones that corresponded to the operations of PMPs:

```
lift :: a → BiPi u a
lift = return

sequ :: (b → a) → BiPi a a → (a → BiPi b b) → BiPi b b
sequ f pa k = comap (Just ∘ f) pa >>= k
```

`lift` corresponds directly to `return`, while `sequ` corresponds to a `bind` where the backward `comap` annotation is enforced. The only difference is that in their restriction the backward annotation will always succeed. Kennedy notes that this forms a monad, missing only our key insight that the addition of profunctors creates the appropriate abstraction for monadic bidirectional programming.

Building from these two core combinators, it is not challenging to reimplement the rest, enabling direct porting of their examples.

As a combinator library, they guarantee round tripping laws by construction; something that in our framework is proved compositionally.

6.3 Round Tripping

Pickler correctness can be encapsulated into two laws:

Definition 6.1. A pickler `p :: BiPi u v` is *backward round tripping* when

$$\texttt{unpickle}\ u = \texttt{Just}\ u' \implies \texttt{pickle}\ u' = u$$

Definition 6.2. A pickler `p :: BiPi u v` is *forward round tripping* when

$$\texttt{pickle}\ s = \texttt{Just}\ u \implies \texttt{unpickle}\ u = \texttt{Just}\ s$$

Unpacking, then repacking something should not change it; and if a serialisation was successful, unpacking it should obtain the original value.

As with biparsers, these laws cannot be proven compositionally as they are: the backward direction must be broken down into a weakened form with a pure projection; and the forward must also be weakened then instead be proven using quasicompositionality. We will look at each in turn.

The backward round tripping law has to be weakened to support the monadic bind operation, removing reliance on a connection between the two type parameters (`u` and `v`). A connection which is recovered later via purification.

Definition 6.3. A pickler `p :: BiPi u v` is *weak backward round tripping* when for all `x :: u` and `s :: String` and `f :: String → String`, we have:

$$\texttt{pick p x = Just (v, f)} \implies \texttt{unpick p (appEndo f s) = Just (v, s)}$$

The connection between `u` and `v` is broken by having `pick` output `v`, representing the value it has pickled (`x`), but of the correct type. `v` is then used to verify the behaviour of `unpick`. This weakened property is similar to that of biparsers, where `s` is the remaining string, except this time `appEndo` is used instead of `++`. `appEndo :: Endo a → a → a` unpacks the `Endo` type, which is a **newtype** wrapper around endomorphisms in Haskell allowing them to have a monoid instance.

Theorem 6.4. Weak backward round tripping is a compositional property.

To recover full backward round tripping, all that remains is the definition of `proj` to show that `BiPi` is a purifiable PMP. Since all that has changed between biparsers and picklers is the writer's monoid, the definition uses the same structure.

Proposition 6.5. `BiPi` is a *purifiable* PMP (Definition 5.16), with `proj` defined as:

```
proj :: BiPi u v → (u → Maybe v)
proj b u = fmap fst (pick b u)
```

Thus backward round tripping for picklers is compositional.

When it comes to the forward direction, the round tripping cannot be proven compositionally; instead quasicompositionally is needed. To prove quasicompositionality the round tripping law again needs weakened:

Definition 6.6. A pickler `p :: BiPi u v` is *weak forward round tripping* when for all `s1 :: String` and `s2 :: String`, we have:

$$\texttt{unpick p (s1 ++ s2) = Just (x, s2)} \implies \texttt{pickle p x = Just s1}$$

If some value is unpacked, leaving some remainder (`s2`), repacking that value should yield the prefix of the original string (`s1`).

Theorem 6.7. Weak forward round tripping is a quasicompositional property.

Thus, using our reasoning framework, round tripping for picklers can be proven compositionally.

7 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 that we introduced in Section 1:

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

7.1 Construction

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 PMPs `Fwd m` and `Bwd n` for monads `m` and `n`. For lenses we are in the same position: the forward direction, `get`, is already a monad—the reader monad. The backward direction, `put`, is not a monad since it is contravariant in its parameter; the same situation as printers. The same approach of "monadisation" used for parsers and printers yields the following new data type for lenses:

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

The result of `put` is paired with a covariant parameter `v` (the result type of `get`) 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`. Note that this is essentially the state monad. This definition can be structured as a PMP via a pair of `Fwd` and `Bwd` constructions:

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

Thus by the results of Section 4, we now have a PMP 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" [Mu et al. 2004], 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 [Foster et al. 2007] or dynamically [Mu et al. 2004]. To remedy this, we capture both partiality of `get` with `Maybe`, and a predicate `(s → Bool)` on sources in `put` for additional dynamic checking. This is provided by the following `Fwd` and `Bwd` PMPs:

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

Going forward, *getting* a view `v` from a source `s` may fail if there is no view for the current source. Going backward, *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, § 4.2, p. 13). 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 bit 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:

```
put l x s = Just ((_, s'), p)  ∧  p s'  ⟹  get l s' = Just x      (Acceptability)
      get l s = Just x     ⟹     put l x s = Just ((_, s), _)      (Consistency)
```

Acceptability and Consistency [Matsuda and Wang 2015] are backward and forward round tripping respectively, and collectively, the two laws define *well-behavedness* [Bancilhon and Spyratos 1981; Foster et al. 2007; Hegner 1990]. 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 ⊆ u` when the above properties only hold on elements `x` of `P`.

For every source type `s`, the lens type `L s` is automatically a PMP by its definition as the pairing of `Fwd` and `Bwd` (Section 4.1), and the following instance of `Monoid` to satisfy the requirements of the writer monad, which is the (→) monoid composed with the `And` monoid:

```
instance Monoid (s → Bool) where
  mempty = λ_ → True
  mappend h j = λs → h s ∧ j s
```

7.2 A Small Example: Key-Value Maps

A simple lens example that 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-focused 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 looks up the key `k` in a map, producing an optional `Value` if found. The `put` component inserts a value for key `k`. When the key already exists, `put` overwrites its associated value.

Thanks 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)
  xs ← comap tailM (atKeys ks)
  return (x:xs)
```

7.3 A Bigger Example: Trees

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

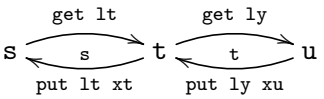
```
data Tree = Leaf | Node Tree Int Tree deriving Eq
```

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 backward direction will then allow a tree to be updated with a new right spine (represented as a list of integers), which may produce a larger source tree.

We start by defining a 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 = do
    t ← get lt s
    ((y, xt), q') ← put ly xu t
    ((_, s'), p') ← put lt xt s
    let check s' =
      p' s' ∧ any q' (get lt s')
    Just ((y, s'), check)
```

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 `put'` in (>>>), for the composite backward direction to succeed, the returned intermediate store `xt` must be consistent (free of conflict) as checked by `q'`.

We define two primitive lenses: `rootL` and `rightL` for the root and right child of a tree. 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 the use of `Maybe` inside the definition of `L`: internally `L` uses `Maybe` to represent failure, here at the top-level we are using it to explicitly indicate presence of absence of a label without failing.

```
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''
```

The `rightL` lens accesses the right child of a tree. If the source tree is a `Leaf`, this lens fails rather than return a `Maybe` view. In both lenses `rootL` and `rightL`, the `put` operations return predicates that check that the view of a (possibly updated) source is equal to the view provided to `put`.

```
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')
```

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
```

The lens `spineL`, interpreted as a `get`, first views the root of the source tree through `rootL` as `hd`, and recurses if the root is a non-empty node (with label `n`), 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)))
```

7.4 Round Tripping

To apply the reasoning framework of Section 5, to Acceptability and Consistency we must consider a couple of things to ensure compositionality. We first weaken backward round tripping (Acceptability) to be compositional. Here, our first consideration is that we can no longer reply on the lens being aligned. This means that we cannot compare the input to `put` with the output of `get`

as they will have different types. Instead, we must make use of the output of `put`, which has the correct type. Next, we must consider how the predicate that tests for conflicts under composition. By sequentially composing lenses such as in `l ≫= k`, the output source `sl` of `put l` will be further modified by `put (k x)`, so this round tripping property must constrain all potential modifications of `sl`. In fact, the predicate `p` ensures exactly that the view `get l` has not changed and is still `x`. It is not even necessary to refer to the intermediate source `sl`, which is just one source for which we expect `p` to be `True`. These changes give us our weak backward round tripping:

Definition 7.1. A lens `l :: L s u v` is *weak backward round tripping* if for all `x :: u`, `y :: v`, for all sources `s`, `s'`, and for all `p :: s → Bool`, we have:

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

Theorem 7.2. Weak backward round tripping is a compositional property.

Again, we complement this weakened version of round tripping with the notion of purification to recover the regular backward round tripping property.

Proposition 7.3. Our lens type `L` is a *purifiable* PMP (Definition 5.16), with a family of pure projections `proj s` indexed by a source `s`, defined as:

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

Theorem 7.4. If a lens `l :: L s u u` is weak backward round tripping and projects to the identity on some subset `P ⊆ u` (*i.e.*, `x ∈ P ⇒ proj s l x = Just x`, *cf.* Definition 5.18) then `l` is also backward round tripping on all `x ∈ P`.

Proof. Assume `put l x s = Just ((y, s'), p) ∧ p s' = True`. By the identity projection assumption, `x = y`. We can now apply weak backward round tripping to deduce `get l s' = Just x`. □

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

Proposition 7.5. The 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) s' = True
```

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

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

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

Proposition 7.6. The lens `atKey k` projects to the identity: `proj z (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
```

□

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

Corollary 7.8. 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 (Consistency), follows a similar story. We first restate it as a quasicompositional property.

Definition 7.9. A lens `l :: L s u v` is *weak forward round tripping* if for all `x :: u`, `y :: v`, for all sources `s`, `s'`, and for all `p :: s → Bool`, we have:

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

Theorem 7.10. Weak forward round tripping is a quasicompositional property.

Along with identity projection, this gives the forward round tripping property.

Theorem 7.11. If a lens `l` is weak forward round tripping and projects to the identity on some subset `P` (*i.e.*, `x ∈ P ⇒ proj s l x = Just x`) then `l` is also forward round tripping on `P`.

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

Proposition 7.12. For all `ks`, the lens `atKeys ks :: L Src [Value] [Value]` is forward round tripping on lists of length `length ks`.

8 Monadic bidirectional programming for generators

This section captures the novel notion of *bidirectional generators* (*bigenerators*). These expand the applicability of bidirectional programming to random generators in property-based testing frameworks like *QuickCheck* [Claessen and Hughes 2000].

8.1 Construction

The forward direction generates values conforming to a specification; the backward direction checks whether values conform to a predicate. We capture the two together via our PMP pair as `G`:

```
type G = (Fwd Gen):*:(Bwd Maybe)
 -- ... with destructors 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
```

A bigenerator is value of type `G u v`, representing a subset of `v`. The forward direction of the bigenerator is given by `generate` that returns a generator of type `Gen v`. The backward direction is given by `check` that returns a partial function of type `u → Maybe v`. The application `check g` defines a predicate on `u` that is true if and only if `check g u` returns `Just x` for some value `x`. This predicate can be extracted from the backward direction using the function `toPredicate`:

```
toPredicate :: G u v → u → Bool
toPredicate g x = isJust (check g x) where
```

```
isJust (Just _) = True
isJust Nothing = False
```

The bigenerator type `G` is automatically a PMP due to our construction (§4). Thus, monad and profunctor instances come for free, modulo (un)wrapping of constructors.

8.2 Examples

This section provides code examples for our notion of *bigenerators* (*bidirectional generators*) extending random generators in property-based testing frameworks like *QuickCheck* [Claessen and Hughes 2000] 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 definition above. We can similarly write a *checker* for binary search trees as a predicate.

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

```
checkBST :: Int → Int → Tree → Bool
checkBST lo hi Leaf         = True
checkBST lo hi (Node l n r) =
     lo ⩽ n
   ∧ n  ⩽ hi
   ∧ checkBST lo n l
   ∧ checkBST n hi 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)
```

The generator `genBST` above depends on two primitives: `genBool` to decide whether to generate a leaf or a node, and `choose` to generate a value for a node. We wrap them as the following bidirectional generators, `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 :: Tree → Maybe Tree
nodeLeft (Node l _ _) = Just l
nodeLeft _ = Nothing

nodeRight :: Tree → Maybe Tree
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.

8.3 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 and the predicate of a bigenerator `g` must match. 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

a predicate (e.g. a Haskell function `isBST :: Tree → Bool`), but completeness determines the potential adequacy of testing, a generator should be able to generate all values in the test space. In other words, overfitting the test space can be fixed by filtering, but underfitting is an issue.

Definition 8.1. A bigenerator `g :: G u u` is *complete* (backward round tripping) when `toPredicate g x = True` implies `x ∈ generate g`.

Definition 8.2. 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 purification property.

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

Definition 8.3. A bigenerator `g :: G u v` is *weak-complete* when

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

Theorem 8.4. 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 8.5. A bigenerator `g :: G u u` is complete if it is weak-complete and its checker satisfies a pure projection property: `check g x = Just x' ⇒ x = x'`(that is, if a succesful bigenerator g "projects to the identity" in the terminology of Definition 5.17).

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 no decomposition of soundness into a quasicompositional property and the pure projection property. The following (counter)example shows why:

Example 8.6. Let `bool :: G Bool Bool` be a random uniform bigenerator of booleans, then consider two contextualisation of `bool`:

```
comap isTrue bool                (4)
comap isTrue (return True)       (5)
```

where `isTrue True = Just True` and `isTrue False = Nothing`.

Both the above satisfy any quasicompositional property satisfied by `bool`, and both have the same pure projection `isTrue`, and yet (4) is unsound—in the forwards direction it can generate `False`, which is rejected by `isTrue`, returning `Nothing`—whilst (5) is sound.

To reason about soundness, we must look beyond quasicompositionality. All of the monadic bidirectional programs presented in this paper follow a regular structure: every (≫=) is associated with a `comap`, and the function argument of `comap` is the sagittal inverse of the function argument of (≫=). In other words, we use (≫=) to apply mappings in the "forward" direction, and `comap` to apply mappings "backwards", and these mappings should be inverse of each other. In the previous sections, we used the notion of pure projection to express this inverse relation, and the counterexample above shows that that approach is not suitable to reason about completeness of bigenerators. Instead, we introduce the following notion of *idiomatic compositionality* which constrains (≫=) and `comap` to use an injective function and its inverse.

Definition 8.7. Let `P` be a PMP. A property $\mathcal{R}_{\mathtt{u}} \subseteq$ `P u u` indexed by a type `u` is *idomatically compositional* if the following holds, for all `x :: u`, `p :: P u u`, `k :: u → P v v`, `f :: v → Maybe u`:

$$\mathcal{R}_{\mathtt{u}}(\mathtt{return\ x}) \qquad \text{(icomp-return)}$$

if `k` is an injective arrow with inverse `f`,

$$(\mathcal{R}_{\mathtt{u}}\mathtt{p}) \ \wedge \ (\forall \mathtt{x}.\ \mathcal{R}_{\mathtt{v}}(\mathtt{k\ x})) \implies \mathcal{R}_{\mathtt{v}}(\mathtt{comap\ f\ p} \gg\!= \mathtt{k}) \qquad \text{(icomp-bind)}$$

Like quasicompositionality, it requires $(\gg\!=)$ to be applied to an injective arrow `k`, but idiomatic compositionality adds the constraint that the inverse of `k` appears as the argument of the corresponding `comap`.

Theorem 8.8. Quasicompositionality implies idiomatic compositionality.

One notable difference is that idiomatic compositionality only considers computations of type `P u u`, where the "input" type and the "output" type are the same `u`. That allows the property $\mathcal{R}$ to encode a notion of round trip between those types, which was previously the role of the pure projection property. In contrast, compositionality and quasicompositionality applied to properties of computations of type `P u v`, where the "input" type `u` and the "output" type `v` may vary independently, making such a notion of round trip inexpressible.

We can formulate a stronger variant of soundness that is idiomatically compositional.

Definition 8.9. A bigenerator `g :: G u u` is *strongly sound* (strongly forward round tripping) if for all `x :: u`, `x ∈ generate g` implies that `check g x = Just x`.

Whereas soundness relates the generator `generate g` to the predicate `toPredicate g`, strong soundness relates the generator to the *partial function* `check g` from which the predicate is derived. The partial function `check g` must behave as the identity function (`Just`) on the domain of the generator `generate g`.

Theorem 8.10. Strong soundness implies soundness.

Theorem 8.11. Strong soundness is idiomatically compositional.

Idiomatic compositionality means that proving that a bigenerator satisfies strong soundness amounts to checking that all uses of $(\gg\!=)$ are paired with `comap` with an semi-inverse relation between their function arguments, and that the primitives are strongly sound.

Lemma 8.12. `bool` and `inRange` are strongly sound.

Corollary 8.13. `bst` is strongly sound.

Hence, `bst` is a fully round tripping generator, *i.e.*, sound and complete.

Reflective Generators. Reflective generators [Goldstein et al. 2023] build upon our idea of bigenerators, combining them with free generators [Goldstein and Pierce 2022] and getting more out of the backward direction. Free generators consider generators to be parsers of randomness, which can be complemented well by a backward direction that unpacks the choices that led to a value. Thus combining free generators with PMPs gives rise to many appealing applications including shrinking and mutation.

9 Monadic Bidirectional Logic Programming

Bigenerators open the door to a whole new world of bidirectional fun. Where reflective generators play with the backward direction, we now explore the forward, as there is no need to limit it to just the `Gen` monad. Why not substitute in lists and play with non-determinism and backtracking,

entering the realm of logic programming? For soundness and completeness, any monad that behaves like a set will do, and that is the core idea that helps explore the limits of our new notion of bigenerators.

9.1 Construction

We generalise the type `G` (shown below for comparison) from `Gen` in the forward direction to any monad, yielding the `GG` data type:

```
type G     u v =              (Fwd Gen:*:Bwd Maybe) u v
type GG m u v = Monad m ⇒ (Fwd m  :*:Bwd Maybe) u v
```

The constructors and destructors also remain the same, just with a more general type:

```
 -- constructors:
mkGG :: Monad m ⇒ m v → (u → Maybe v) → GG m u v
 -- destructors:
generate    :: Monad m ⇒ GG m u v → m v
check       :: Monad m ⇒ GG m u v → u → Maybe v
toPredicate :: Monad m ⇒ GG m u v → u → Bool
```

Similar to bigenerators, we provide a smart constructor which accepts a predicate (`v → Bool`) which is then internally turned into a function of type (`v → Maybe v`) for `mkGG`:

```
mkAlignedGG :: Monad m ⇒ m v → (v → Bool) → GG m v v
mkAlignedGG gen chk = mkGG gen (λy → if chk y then Just y else Nothing)
```

We refer to these as *aligned generalised bigenerators* as the view and pre-view types are the same.

Intuitively, the forward direction (`generate`) can still be understood as generating values fulfilling a predicate, and the backward direction (`check`) as a checker for that predicate. The only change is how the generated values are collected, notions of correctness are the same. It is still desirable for generalised generators to be *sound* and *complete*. In proving these, the same concepts as before arise: completeness can be shown through weak completeness and a pure projection, and soundness must be strengthened and proven using idiomatic compositionality.

The bigenerator of Section 8 can be replicated trivially by specialising the monad in the above construction to `Gen`. We instead explore the expressive power of programs written in this generalised bigenerator form, capturing logic programming problems involving search. Section 10 shows that a generalisation to the `PartialProfunctor` notion allows further power for logic programming.

9.2 Choice and Logic Programming

The `MonadPlus` type class provides an interface generalising non-determinstic computation:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

In this context, the monad operation `return :: a → m a` represents the creation of a determinstic computation, with a single choice, `mzero` represents an empty choice (perhaps a kind of failure) and `mplus` captures a non-deterministic choice. From this interface, examples from logic programming can be captured by keeping the monad general, but restricted to `MonadPlus` instances, so that the search strategy can be later selected by specialising the monad in the fashion of "Transforming Functional Logic Programs into Monadic Functional Programs" [Braßel et al. 2010].

By specialising `GG` to monads that are instances of `MonadPlus` we can replicate the examples of Braßel et al. [2010] but generalised to *bidirectional* logic programs. For example, a nondeterministic choice of a boolean can be implemented as an atomic `GG` computation:

```
trueOrFalse :: MonadPlus m ⇒ GG m Bool Bool
trueOrFalse = mkGG
  (mplus (pure True) (pure False))
  (const (Just True))
```

The forward 'generate' direction takes advantage of the `MonadPlus` interface to lift the two options into the monad via `mplus`. This is the approach taken by Braßel et al. [2010], motivated by expressing the program `True ? False` from the functional logic language Curry [Hanus et al. 1995]. The backwards 'check' direction needs to take a `Bool` value and assert that it is either `True` or `False`. Since this is something that is always the case due to the Haskell type system and definition of `Bool`, the check direction can be trivially defined as always returning `Just True`.

Our monadic bidirectional interface can then be leveraged to compose `trueOrFalse` into more complex examples. For example, to generate/classify lists with a single boolean, we can write:

```
oneBool :: MonadPlus m ⇒ GG m [Bool] [Bool]
oneBool = do
  b ← comap singleton trueOrFalse
  return [b]
```

Here, `return` wraps the resulting boolean into a singleton list, and `comap` adjusts its context from `Bool` to `[Bool]` using `singleton :: [a] → Maybe a` to classify singleton list solutions, which returns a `Just` when there is only one element in the list, wrapping that single element.

We build from here to more complex logic programming problems involving search.

N-queens. The classic N-queens puzzle involves placing N queen pieces on a chessboard such that they cannot take each other. For example, placing two queens next to each other (horizontally, vertically, or diagonally) is invalid, but placing one a knight's move away from the other is valid. The N-queens problem lends itself well to this framework, showcasing how the monadic style allows bidirectional solutions to be written in idiomatic Haskell, featuring pattern matching and recursion, and without committing to the choice of underlying search monad.

We define the top level as `nQueens` below using the monadic interface, `MonadPlus`, and `MonadFail`:

```
nQueens :: (MonadPlus m, MonadFail m)
        ⇒ Int → GG m [Pos] [Pos]
nQueens 0 = return []
nQueens n
  | n > 8 = reject
  | n < 0 = reject
  | otherwise = do
    xs ← comap safeTail (nQueens (n − 1))
    x  ← comap safeHead (safePlacing xs)
    return (x:xs)
```

```
-- Chess positions as integer pairs
type Pos = (Int, Int)

reject :: MonadFail m ⇒ GG m a a
reject =
  mkAlignedGG
    (fail "No")    -- No results
    (const False) -- Always fail
```

When there are no queens to place (first case), the job is easy: `nQueens` can just return the empty list. We also reject more than eight queens (they won't fit on the board in a civilised manner), or a negative number of queens, handled by the pattern guards in the next case. When it comes to

the inductive case (`otherwise` branch), recursion can calculate the solution to $(n-1)$-queens, reducing the problem to calculating where the next queen can be placed by `safePlacing` (below), where non-monadic functions are lifted to the `GG` type using `mkAlignedGG`:

```
safePlacing :: MonadPlus m
  ⇒ [Pos] -- Existing queens positions
  → GG m Pos Pos
safePlacing qs =
  mkAlignedGG
    (choose $ filter noConflicts boardPos)
    (λp → isBoardPos p ∧ noConflicts p)
  where
    -- Checks for conflict with existing queens
    noConflicts p = all (¬ ∘ conflict p) qs

choose :: MonadPlus m ⇒ [a] → m a
choose =
  foldr (λx → mplus (return x)) mzero
```

```
boardPos :: [Pos]
boardPos = [(x, y) | x ← [0 . . 7]
                   , y ← [0 . . 7]]

isBoardPos :: Pos → Bool
isBoardPos (x, y) = 0 ⩽ x ∧ x ⩽ 7
                  ∧ 0 ⩽ y ∧ y ⩽ 7

conflict :: Pos → Pos → Bool
conflict (x1, y1) (x2, y2)
 -- horiz. conflict
   | y1 ≡ y2 = True

 -- vert. conflict
   | x1 ≡ x2 = True

 -- diagonal conflict
   | otherwise =
     abs (x1 − x2) ≡ abs (y1 − y2)
```

The `safePlacing` bigenerator, in the check direction, checks whether the position fits on the chessboard and does not conflict with queens already on the board (`noConflicts`). The generate direction filters all possible positions (`boardPos`) for those which are conflict-free and then uses `choose`, to abstract this list to `MonadPlus` (providing operations `mplus` and `mzero`) to choose between these possibilities.

One reasonable specialisation of the monad here is to lists, allowing the solutions to be enumerated in 'row order'. For example:

```
allSolutions :: [[Pos]]
allSolutions = generate (nQueens 8)

checkOneSolution :: Maybe [Pos]
checkOneSolution = check (nQueens 8 :: GG [] [Pos] [Pos])
  [(2, 7), (3, 2), (6, 1), (7, 4), (4, 6), (5, 3), (1, 5), (0, 0)] -- one solution
```

The N-queens problem can be solved in this way (by induction on the problem size), because N-queens is *antitone* with respect to the `check` function and a subproblem relation. That is, a potential solution will only be valid (according to `check`) if its subsolution is valid:

$$\frac{\texttt{x subproblem of y}}{\texttt{check y} \implies \texttt{check x}}$$

This means that the 2-queens solution must be correct to form a correct 3-queens solution, and so on. This makes the solution straightforward to build by monadic sequential composition, because the solution can be formulated inductively by (monadically) composing (via the **do** notation) the subproblem solution with a way to find the current solution, as seen above in `nQueens`.

However, there are many problems where the relationship between subproblems and predicates is not antitone. To solve this, we introduce the notion of 'recoverability' where a configuration which is a non-solution can be 'recovered' into a correct solution, providing a bidirectional dual to backtracking. We demonstrate this via the traditional *river crossing problem*.

River Crossing. On one side of a river is a fox, a chicken, some grain, and a human who must get everyone across the river by boat. However, there is a catch: the boat is only big enough for two, one of which must be the human to steer, and the fox cannot be left alone with the chicken, nor the chicken be left alone with the grain, lest someone get eaten.

A little more work is required to solve this problem, because unlike N-queens, a bad state can be recovered from by adding extra information. For example, if there is a bank with the chicken and the fox, which does not look good for the chicken, this can be recovered by adding the human to keep the peace. In contrast, if two queens are in conflict, no addition to the board can resolve it.

To implement a solution to this puzzle, the backwards monad needs to support two levels of failure: *recoverable* and *irrecoverable*. Recoverable failures are like the example with the chicken and the fox. In their current state, they will fail, but this can be remedied. Recoverability is represented using the `State` monad, with a boolean indicating whether the current configuration is *unsafe*, meaning that either the fox is left unattended with the chicken, or the chicken with the grain. Both situations may be remediated by adding a human to the configuration. Irrecoverable failures are when the rules of the puzzle are violated, e.g., three characters on the boat. These are represented, as before, with `Maybe`. Adding the new `State` layer for recoverable failure creates *generalised generators with recoverability* (`GGR`s):

```
newtype GGR m u v = GGR ((:*:) (Fwd m) (Bwd (StateT Bool Maybe)) u v)
```

The forward (generate) direction is still a general monad, but the backward (check) direction now has an extra layer of recoverable failure using the `StateT` monad transformer over `Maybe`, giving this monad the ability to keep track of the current validity of a solution as state and short circuit to `Nothing` if the rules of the puzzle are broken (irrecoverable failure).

Setting the monad `m` to lists can be used to solve the river crossing puzzle, but leads to an explosion of solution permutations as the riverbank `[Human, Chicken]` is considered different to `[Chicken, Human]`. In this puzzle, only who is on the bank matters, not their order, suggesting the need for a data type that more closely corresponds to the desired puzzle semantics. The order-agnostic cousin of the list monad, `Set`, seems like the natural solution. However, in Haskell, this leads to an implementational detour because `Set` is not a Haskell monad (its type constraint `Ord` gets in the way of creating a `Monad` instance; essentially `Set` is a monad only on a subcategory of types with an ordering). Luckily, previous work on *relative monads* [Altenkirch et al. 2010] and their implementation in Haskell [Orchard and Mycroft 2012] addresses this problem, allowing for an implementation using `Set`. We elide some of the details here which entails using the `RebindableSyntax` extension to allow the **do**-notation in Haskell to use a relative monad type class `RMonad` instead of the standard `Monad` type class; the programming pattern remains the same.

In terms of scale, solving the river crossing puzzle is a step up from N-queens. The style of programming remains the same, there is still a combination of atomic and combined PMPs, with the former lifting Haskell functions to perform the puzzle logic, and the latter combining the atomic results into the correct form of solution. We capture the problem with a data type `Actor` of the characters in the problem, a relation `compatible` of which actors can be safely on the same riverside, and a designation of the riverside by `Side` which can be flipped by `rowing`:

```
data Actor = Human | Fox | Chicken | Grain
  deriving (Show, Eq, Ord)

compatible :: Actor → Actor → Bool
compatible Fox     Chicken = False
compatible Chicken Grain   = False
compatible Chicken Fox     = False
compatible Grain   Chicken = False
compatible _       _       = True
```

```
data Side = L | R
  deriving (Show, Eq, Ord)

row :: Side → Side
row L = R
row R = L

characters :: Set Actor
characters = S.fromList
  [Human, Chicken, Fox, Grain]
```

Given a set of actors $\mathcal{S}$, we can generate and classify elements of the equivalence class of compatible characters in $\mathcal{S}$ (as defined by `compatible`) as a primitive `GGR` computation constructed via `mkRecoverable :: RMonad m ⇒ m v → (v → Bool) → GGR m v v` (a smart constructor similar to that of bigenerators used before): function:

```
compat :: Set Actor → GGR Set Actor Actor
compat xs = mkRecoverable gen check
  where
    check c = (all (compatible c) xs ∨ Human ∈ xs) ∧ ¬ (elem c xs)
    gen | Human ∈ xs = characters S.\\ xs -- add anyone new
        | otherwise  =                   -- add someone non-conflicting
          foldl (λn e → S.filter (compatible e) n) (characters S.\\ xs) xs
```

This is the analogue of `safePlacing` from `nQueens`. We can compose `compat` iteratively to generate and classify sets of actors that are safe together on a single side via `safeSide` which resembles closely the logical structure of `nQueens` function which similarly generated valid states:

```
safeSide :: Int → GGR Set (Set Actor) (Set Actor)
safeSide 0 = return S.empty
safeSide n -- number of actors on the side
   | n > 4 = reject
   | n < 0 = reject
   | otherwise = do
     xs ← comap tailS (safeSide (n − 1))
     x  ← comap headS (compat xs)
     return (S.insert x xs)
```

Just like `nQueens`, `safeSide` branches on how many actors are required (the first parameter). When there are no actors, it can simply return the empty set. If there are negative actors or more than four, then `safeSide` results in an irrecoverable failure via `reject` (the generate direction of which is just the emptyset `S.empty` but the check direction denotes an irrecoverable failure via `Nothing`). Given a valid size, `safeSide` then inductively creates its solution in the same way as `nQueens`, using `Set` versions of `(:)`, `head`, and `tail`, and using the primitive `compat` for the next compatible element. The inductive step `comap tailS (safeSide (n − 1))` may return an invalid state (e.g., if the human is not present on the side), marked with `False`. However this invalidity is recoverable in the next line which adds a human back in.

From `safeSide` we can generate states of the entire puzzle (not just a single side of the river), captured by the `PuzzleState` data type:

```
data PuzzleState = PuzzleState
  {leftBank     :: Set Actor
  , boat        :: Set Actor
  , rightBank   :: Set Actor
  , boatLocation :: Side
  } deriving (Show, Eq, Ord)
```

A transition function `reachableStates :: PuzzleState → Set PuzzleState` (elided for brevity) takes a puzzle state and computes the set of next possible states, these will satisfy the rules of the puzzle (e.g. no more than three on the boat), but may lead to a failing solution (e.g. someone getting eaten). From `reachableStates` we can make a simple bidirectional computation that generates reachable states or classifies those in a set as to whether they follow the rules of the puzzle (where violation is captured by the `Nothing` in the backward direction):

```
reachableState :: PuzzleState → GGR Set PuzzleState PuzzleState
reachableState current
   = mkIrrecoverable
    (reachableStates current)
    (∈ reachableStates current)
```

We can now create the logic bigenerater `safeState` for safe states that obey the rules of the puzzle and won't lead to a failing solution, by using the **do**-notation interface to cross-reference reachable states generated by `reachableStates` with `safeSide`'s notion of what a safe side is:

```
safeState :: PuzzleState → GGR Set PuzzleState PuzzleState
safeState current = do
  (PuzzleState l b r s) ← reachableState current
  l' ← comap (Just ∘ leftBank) (safeSide (S.size l))
  r' ← comap (Just ∘ rightBank) (safeSide (S.size r))
  if l ≡ l' ∧ r ≡ r'
    then return (PuzzleState l b r s)
    else mzeroR
```

For each of these reachable states generated by `reachableState`, `safeState` ensures that the left `l` and right riverbanks `r` are in the set of safe banks given by `safeSide`. Upon a valid and safe state, the state is added, otherwise the program moves on. The execution of the reasoning is achieved by using a `MonadPlus` like interface, adapted for relative monads like `Set`.

With a means to generate the next puzzle state from a current one, all that remains is gathering a sequence of such valid steps into a solution for the river crossing puzzle. Defined as `riverCrossing` below, it first provides the initial state (everyone on the left bank with the boat) and iteratively uses and collects the results of `safeState` until the end state (everyone on the other bank) is reached. Our solution additionally prevents the revisiting of states to avoid solutions that loop:

```
riverCrossing :: GGR Set [PuzzleState] [PuzzleState]
riverCrossing = riverCrossing' S.empty startState
  where
    riverCrossing'
       ::  Set PuzzleState -- states that have been
       → PuzzleState       -- current state
       → GGR Set [PuzzleState] [PuzzleState]
```

```
riverCrossing' ss s = do
  x ← comap headM (safeState s)
  case complete x of
    True → return [x]
    False → case x `S.member` (S.insert s ss) of
      True → mzeroR -- we are not allowing repeated states
      False → do
        xs ← comap tailM (riverCrossing' (S.insert s ss) x)
        return (x:xs)
```

A version of `MonadPlus` for the relative monads interface (with operation `mzeroR`) is used to cull failed solution branches, and the expressive **do**-notation allows for easy composition of the correct solution, using pure Haskell functions to perform puzzle logic, such as establishing whether the solution is `complete`.

Evaluating (`generate riverCrossing`) produces a set containing both non-looping solutions to the puzzle: one where the fox gets taken before the grain, and one where the grain gets taken before the fox, always with the chicken sticking with the human to avoid it getting eaten/eating. These correct solutions can also be verified in the other direction.

The river crossing puzzle can be characterised as antitone with respect to recoverability.

$$\frac{x \text{ subproblem of } y}{\texttt{Recoverable}\ y \implies \texttt{Recoverable}\ x}$$

When x is a subproblem of y, if y can be recovered (e.g. a bank containing all the animals), then it must have been the case that x was recoverable (in either case the human can be added). However if y is not recoverable (e.g. having three people on the boat) we do not know if x was recoverable: additions that created y from x could have invalidated the solution (adding third person to the boat), or things could already have gone wrong (two animals were on the boat).

The notation of recoverability complicates proof of the round tripping properties for this example. However, we can still achieve a notion of completeness as long as the extra layer of failure is accounted for. A naïve first attempt adapts weak completeness of bigenerators `BiGen` (replayed as (6) below) to the new interface for recoverability, labelled as (7):

$$\texttt{(check g x = Just y)} \implies \texttt{y} \in \texttt{generate g} \tag{6}$$

$$\texttt{(runStateT (check g x) True = Just (y, True))} \implies \texttt{y} \in \texttt{generate g} \tag{7}$$

If the forward monad is restricted to be `Foldable` (allowing the use of $\in$), we have a well-typed generalised bigenerator, which may be compositional, but does not properly account for the state, opening the door for a counterexample to slip in. The counterexample arises because the antecedent of completeness above asserts that the state must be `True` after running `g`. A `GGR` that is not complete can be created that satisfies the round trip law by simply ending with a state of `False` and contradicting the antecedent. To combat this, the check that the state ends as `True`, indicating that no recoverable failures have happened, has been moved to the consequent of the law:

$$\texttt{(runStateT (check g x) True = Just (y, ult))} \implies \texttt{ult} \wedge \texttt{y} \in \texttt{generate g}$$

This ensures that the round trip cannot be upheld vacuously, but has the downside that it doesn't provide any insight into what a failing computation should do. Frustratingly, this prevents the proof that river crossing upholds this law from being given compositionally, as some of river crossing's atomic components, such as `compat`, need to be able to fail in a recoverable manner. River crossing

can still be proven correct via other methods (and testing suggests that it is), but there is certainly more to explore in this new recoverable frontier.

In summary, while this puzzle is still solvable in our framework, the PMP and round tripping laws are complicated by recoverability, which undermines compositionality.

10 Generalising partial profunctors to right monadic actions

A useful generalisation to the key structures of our framework presents itself in the application to logic programming. Recall that, so far, our PMPs also have the `PartialProfunctor` structure providing `internaliseMaybe :: p u v → p (Maybe u) v`. This yields a contravariant functor map of the form `comap :: PartialProfunctor p ⇒ (u → Maybe u') → p u' v → p u v`, capturing the idea that a contextualising function for the backward direction may fail. We generalise this notion such that the contextualising map may be within *any monad* as long as the monad can be 'internalised' into the profunctor:

```
class (Profunctor p, Monad m) ⇒ ProfunctorMonadInternalise p m where
  internaliseMonad :: p u v → p (m u) v
```

yielding the contravariant-functor-like map providing extra power to the contextualising function:

```
comap :: (ProfunctorMonadInternalise p m, Profunctor p) ⇒
  (u → m u') → p u' v → p u v
comap f = dimap f id ∘ internaliseMonad
```

More generally, this can be thought of as a *right monad action* operating on a contravariant functor, a concept that is encapsulated into the following class:

```
class (Monad m, Contravariant f) ⇒ MonadCoAction m f where
  coAct :: f u → f (m u)
```

Where `coAct` is both unital and associative:

```
unital = contramap return ∘ coAct ≡ id
assoc  = contramap join ∘ coAct   ≡ coAct ∘ coAct
```

where `contramap` is akin to `fmap`, but for contravariant functor.[2] The `comap` and `contramap` functions resemble each other, with the main difference being their source categories: `comap` maps morphisms of the Kleisli category of the `Maybe` monad, while `contramap` maps morphisms of the base category (i.e., the imagined category `Hask`).

Unpacking the concept of a monad action, the idea of one set 'acting' on another, in its simplest form, is represented by the function `act :: X → Y → Y`, where elements of the set X are mapped to endomorphisms on Y. In its uncurried form, left and right actions are distinguished by what side of the tuple the acting set, X, is on:

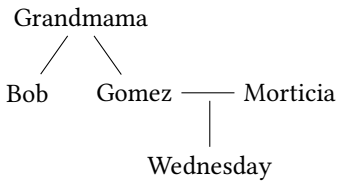
Definition 10.1. A *right action* of a set X on a set Y is defined as follows:

```
actR :: (Y, X) → Y
```

Actions can also respect algebraic structure, for example, a commonly used action is the monoidal action. Here the set X has monoidal structure, and the action can be seen as a homomorphism between the monoid X and the monoid of endomorphisms on Y.

Since monads are monoids in the category of endofunctors, the idea of monoid actions naturally translates to monad actions.

[2]See the Haskell standard library in `Data.Functor.Contravariant`.




```
parents :: [(String, String)]
parents = [
    ("Gomez", "Wednesday")
  , ("Morticia", "Wednesday")
  , ("Grandmama", "Gomez")
  , ("Grandmama", "Bob")]
```

Fig. 1. Addams Family Tree

Definition 10.2. A *monad action* is when a monad m acts upon an endofunctor f either on the left, or on the right as follows:

$$\texttt{mactL} :: \texttt{m (f u)} \to \texttt{f u}$$
$$\texttt{mactR} :: \texttt{f (m u)} \to \texttt{f u}$$

Thus `coAct` corresponds to `mactR`, but acting contravariantly.

Definition 10.3. A *right monad coaction* is a right monad action acting on a contravariant functor:

$$\texttt{coAct} :: \texttt{f u} \to \texttt{f (m u)}$$

Thus, `ProfunctorMonadInternalise` is a right monad coaction on the first (contravariant) parameter of a profunctor.

We can now restate the property of being compositional when generalising a monadic profunctor to have an arbitrary coaction:

Definition 10.4. A property $\mathcal{R}$ is *compositional* with respect to the operations of an M-coacting PMP P if the following conditions hold for all types `u`, `v`, `w`, for all `x :: v`, `p :: P u v`, `k :: v → P u w`, `f :: u' → M u`,

$$(\texttt{return x}) \in \mathcal{R}^{\texttt{u}}_{\texttt{v}} \qquad \text{(comp-return)}$$
$$\left(\texttt{p} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}}\right) \wedge \left(\forall \texttt{v}.\ (\texttt{k v}) \in \mathcal{R}^{\texttt{u}}_{\texttt{w}}\right) \implies (\texttt{p} \gg\!= \texttt{k}) \in \mathcal{R}^{\texttt{u}}_{\texttt{w}} \qquad \text{(comp-bind)}$$
$$\texttt{p} \in \mathcal{R}^{\texttt{u}}_{\texttt{v}} \implies (\texttt{comap f p}) \in \mathcal{R}^{\texttt{u'}}_{\texttt{v}} \qquad \text{(comp-comap)}$$

The only change compared with the previous definition of compositionality (Definition 5.5) is that we have generalised the function `f` applied in `comap` to an arbitrary monad `M`.

We will now consider an example that builds on this generalised ability to 'internalise' monadic computations inside a profunctor, and its associated generalisation of compositional reasoning.

Example: Grandparents. A classic logic programming example asks queries about familial relationships over a database of facts about parentage. Consider the family tree given in Figure 1, which can be encoded by a simple binary relationship parent where parent(x, y) means that x is the parent of y, e.g., parent(Gomez, Wednesday). We consider our bidirectional programming view of queries over this knowledge base and the problem of deriving a notion of *grandparent* from the above data. This puzzle is as simple as looking at a family tree and asking who is whose grandparents. For example, for the Addams family tree (Figure 1), the question could be, "Who is Wednesday's grandparent?". A valid checker would only accept Grandmama as an answer, and a sound and complete enumeration should produce the singleton list containing only Grandmama.

Whilst we previously defined generalised bigenerators `GG` in terms of a forward monad (which could be `Gen`) and backwards `Maybe` monad (to capture checking a predicate), we now generalise `Maybe` to lists, capturing the idea of a list of successes [Wadler 1985]:

```
newtype ND m u v = ND {unND :: (Fwd m:*:Bwd []) u v}

mkND :: Monad m ⇒ m v → (u → [v]) → ND m u v
mkND choices check = ND (Fwd choices:*:Bwd check)
```

Thus we can see in the smart constructor `mkND` that a primitive bidirectional computation has in the backwards direction a map from a pre-view `u` to zero-or-many views `v`.

The deconstructors for the forward and backward direction are projections, given the names:

```
generateND :: Monad m ⇒ ND m u v → m v
generateND = unFwd ∘ pfst ∘ unND

checkND :: Monad m ⇒ ND m u v → u → [v]
checkND = unBwd ∘ psnd ∘ unND
```

This type `ND` admits an instance of `ProfunctorMonadInternalise` (alternatively `MonadCoAction`) where we internalise a list argument in the contravariant position simply by the monadic bind of the list monad in the backwards direction:

```
instance Monad m ⇒ ProfunctorMonadInternalise (ND m) [] where
  internaliseMonad :: ∀u v.ND m u v → ND m [u] v
  internaliseMonad (ND (Fwd py:*:Bwd qy)) = ND (Fwd py:*:Bwd (λx → x ≫= qy))
```

For this particular problem, we instantiate the forwards monad with the list monad.

The Figure 1 relation is captured as a list of string pairs `parents`. From this, `parentOf` provides a non-deterministic computation enumerating the parents of a given person:

```
parentOf :: String → [String]
parentOf person = do
  (parent, child) ← parents
  guard (person ≡ child)
  pure parent
```

We can define a bidirectional version of `parentOf` to generate parents where the backwards direction checks whether elements are in the parent set:

```
biparentOf :: String → ND [] String String
biparentOf n = mkND
  (parentOf n)
  (λs → if s ∈ (parentOf n) then [s] else [])
```

The obvious (but wrong) approach to get a bidirectional computation of grandparents is to compose `biparentOf` with itself twice , i.e.

```
bigrandparentsOfattempt :: String → ND [] String String
bigrandparentsOfattempt c = biparentOf c ≫= biparentOf
```

In the fowards directions (generating), then evaluating:

```
generateND $ bigrandparentsOfattempt "Wednesday"
```

yields the expected result of `"Grandmama"`. However, the backwards direction (checking) does not yield what we expect: `(checkND $ bigrandparentsOfattempt "Wednesday") "Grandmama"` evaluates to `[]`, i.e., it is not calculated that `"Grandmama"` is the grandparent of `"Wednesday"`, contradicting the forward direction; tripping is violated.

The problem is that the first backwards step (the left-most application `biparentOf "Wednesday"`) checks if `"Grandmama"` is an element of the set `parentOf "Wednesday"`, which is false because `parentOf "Wednesday"` generates `"Gomez"` and `"Morticia"`. Thus, the first backwards step yields no results, i.e., `[]`, and so the second backwards step (right-most) does not run.

What is missing is the need to contextualise the first `biParentOf` so that the backwards direction is not given `"Grandmama"` as its input but is given the children of `"Grandmama"`. Thus we contextualise the first use of `biparentOf` via `comap` and the function `children` which is akin to a generator for children, defined similarly to the original `parentOf` generator:

```
bigrandparentsOf ::
  String → ND [] String String
bigrandparentsOf c = do
  parent ← children `comap` biparentOf c
  biparentOf parent
```

```
children :: String → [String]
children parent = do
  (parent', child) ← parents
  guard (parent ≡ parent')
  pure child
```

Evaluating `((checkND $ bigrandparentsOf "Wednesday") "Grandmama")` returns `["Grandmama"]` as expected. The evaluation of the backwards checking direction proceeds by first applying `children "Grandmama"` yielding the `["Gomez", "Bob"]`, each of which is then passed to the backwards direction of `biparentOf "Wednesday"` yielding `["Gomez"]` as the only candidate. Thus the last line of `bigrandparentsOf` is `biparentOf "Gomez"` whose backward direction is not recontextualised and thus considers whether `"Grandmama"` is the the parent of `"Gomez"` which is correct and thus `["Grandmama"]` is the result as expected.

With the previous `Maybe`-based approach in the river crossing and N-queens problems, we could not produce the desired result here as the backwards direction cannot contextualise itself to enough results; a `children :: String → Maybe String` function would not know which child to "pick".

We note that we could also define a variant of the `ND` type here with the logic monad of [Kiselyov et al. 2005], in both directions, providing additional backtracking search facilities.

```
newtype BiLogic u v = BiLogic {unBL :: (Fwd Logic:*:Bwd Logic) u v}
```

All of the above can be reformulated in this profunctorial monad, producing the same results, without the extra indirection of converting `Logic [a] → [a]` in `children`, providing the full power of the `Logic` monad to both directions. This appears to be the most flexible setting for bidirectional logic programming, combining the profunctorial monad approach of this paper's main theme along with the `MonadCoAction` above to subsume all the results of this section.

10.1 Summary

Generalised generators can be used to solve logic puzzles whose solutions are created inductively. This includes N-queens, and more complex puzzles, such as the river crossing puzzle, that include a notion of recoverability, but excludes puzzles of a transitive nature, for example, resolving who is whose grandparent. By generalising our notion of contravariant functoriality from `a → Maybe b` arrows to `a → m b` arrows with a monad coaction, we can capture a much larger class of bidirectional logic problems.

Generalising the novel notion of bigenerators has further extended the boundary of what bidirectional programs can be applied to, a boundary which we have taken care to pin point and characterise, and pushed back at in a monadic way.

11 Discussion and Related Work

Bidirectional transformations are a widely applicable technique used in many domains [Czarnecki et al. 2009]. Among language-based solutions, the lens framework is most influential [Barbosa et al. 2010; Bohannon et al. 2008; Foster et al. 2007, 2010; Pacheco et al. 2014a; Rajkumar et al. 2013]. 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.

11.1 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 primary contribution is the generalization of that pattern to work with monads, a more expressive abstraction than applicative functors. We also introduce notions of alignment and quasicompositionality, which we believe systematically yield simple criteria to guarantee round-trip properties of bidirectional programs. Our framework adapts itself particularly well to types of computations that are only `Applicative`. For example, the following combinator combines two bidirectional transformations into one bidirectional transformation on pairs.

```
pair
  :: (Profmonad p
    , PartialProfunctor p
    , Applicative (p (a, b))
    , Functor     (p (a, b)))
  ⇒ p a a → p b b → p (a, b) (a, b)
pair a b = (, ) <$> comap (Just ∘ fst) a <*> comap (Just ∘ snd) b
```

The `Alternative` typeclass abstracts common forms of backtracking. We also have examples of "alternative profunctors", but do not include them in the paper due to the lack of space. Ostermann and Rendel's developed theory of partial isomorphisms is the basis for their version of a `Functor`. Partial isomorphisms are in fact pairs of pseudo-inverse injections, which our framework allows to be used separately. These injections are also more easily available. In practice, we obtain most of them for free as constructors and field accessors.

As demonstrated above, 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 4 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.

FliPpr [Matsuda and Wang 2013, 2018] is an invertible language that generates a parser from a definition of a 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 the FliPpr work.

11.2 Lenses

More recently, researchers and practitioners have started to realise that some category-theory inspired isomorphisms may turn lenses (which are encapsulated pairs of functions) into representations of single functions. For example, the Haskell lens library [Kmett 2012] uses the so-called *van Laarhoven representation* [van Laarhoven 2009] to replace lens composition with normal functional composition. The Purescript language uses a closely related representation: *profunctor optics* [Pickering et al. 2017], a generalization of lenses to which our work bears a striking resemblance.

More precisely, a *Profunctor optic* [Pickering et al. 2017] 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`.

Promonadic definitions are loosely similar to profunctor optics. By abstracting over primitives, *e.g.* `rootL` and `rightL` in Section 7, 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`. Profunctor optics and our PMPs offer orthogonal composition patterns: profunctor optics can be composed "vertically" using function composition, whereas PMP composition is "horizontal" providing sequential composition. In both cases, composition in the other direction can only be obtained by breaking the abstraction.

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 directly 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 monadically 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.

11.3 Random generators and predicates

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) [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 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 specialised framework for random generators.

And of course, reflective generators [Goldstein et al. 2023], which we mentioned in Section 8, further demonstrate the usefulness of PMPs in this setting.

11.4 Future work

This work presents applications of PMPs to the design and use of embedded domain-specific languages for bidirectional programming. Partiality is pervasive in our development, and the composition patterns captured by quasicompositionality require discipline and are still error prone. We hope to develop tools and methods such as type systems to automatically ensure correct usage of these patterns and guarantee compositional properties.

12 Conclusions

This paper advances the expressive power of bidirectional programming; we showed that the classic bidirectional patterns of parsers/printers and lenses can be restructured in terms of *Partial Monadic Profunctors (PMPs)* to provide sequential composition, with associated reasoning techniques. This 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 bigenerators and their generalisations 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 have developed two bidirectional libraries [Xia 2018], one extending the attoparsec monadic parser combinator library to biparsers and one extending QuickCheck to bigenerators. One area for further work is studying biparsers with *lookahead*. Currently lookahead can be expressed in our extended attoparsec, but understanding its interaction with (quasi)compositional round tripping is further work.

Our framework could be applied in library implementations and in the design of bidirectional languages, with either a codec or PMP basis for the semantics. We have considered some extensions to our notions of compositionality and congruence to include other programming constructs such as sum types. In fact, our work has inspired the creation of tomland[3], a bidirectional TOML parser, which uses PMPs.

However, this is not the final word on sequentially composable BX programs. In 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 formalised, perhaps based on an adjunction model, that captures bidirectionality even more concretely than PMPs.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. Orchard was supported partly by EPSRC grant EP/T013516/1, and also received support through Schmidt Sciences, LLC. Frohlich and Wang were partly supported by EPSRC grant EP/T008911/1.

References

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.

T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In *International Conference on Foundations of Software Science and Computational Structures*, pages 297–311. Springer, 2010.

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 *International Conference on Functional Programming (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 *Symposium on Principles of Database Systems (PODS)*, 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 *Symposium on Principles of Programming Languages (POPL)*, pages 407–419. ACM, 2008.

G.-J. Bottu, G. Karachalias, T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler. Quantified class constraints. In *International Symposium on Haskell (Haskell)*, pages 148–161. ACM, 2017. 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 *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, 2002. ISBN 1-58113-562-9. DOI: 10.1145/566172.566191.

B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *International Workshop on Functional and Constraint Logic Programming*, pages 30–47. Springer, 2010.

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 *International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000. 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. In *International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 260–283. Springer 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 *SSGIP*, pages 1–46, 2010.

H. Goldstein and B. C. Pierce. Parsing randomness. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. DOI: 10.1145/3563291. URL https://doi.org/10.1145/3563291.

H. Goldstein, S. Frohlich, M. Wang, and B. C. Pierce. Reflecting on random generation. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. DOI: 10.1145/3607842. URL https://doi.org/10.1145/3607842.

M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[3]https://kowainik.github.io/posts/2019-01-14-tomland

S. J. Hegner. Foundations of canonical update support for closed database views. In S. Abiteboul and P. C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 1990. ISBN 3-540-53507-1.

N. Jacobson. *Basic algebra I.* Courier Corporation, 2012.

M. P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 97–136, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49270-2.

A. J. Kennedy. Functional pearl pickler combinators. *J. Funct. Program.*, 2004. ISSN 0956-7968.

O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices*, 40(9):192–203, 2005.

E. Kmett. profunctors library, 2011. https://hackage.haskell.org/package/profunctors-5.6.2, accessed January 2021.

E. Kmett. lens library, 2012. https://hackage.haskell.org/package/lens-5.1, accessed January 2021.

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 *Symposium on Principles of Programming Languages (POPL)*, pages 114–129. ACM, 2017.

K. Matsuda and M. Wang. FliPpr: A Prettier Invertible Printing System. In *European Symposium on Programming (ESOP)*, pages 101–120. Springer, 2013.

K. Matsuda and M. Wang. Applicative bidirectional programming with lenses. In *International Conference on Functional Programming (ICFP)*, pages 62–74. ACM, 2015. DOI: 10.1145/2784731.2784750.

K. Matsuda and M. Wang. Embedding Invertible Languages with Binders: A Case of the FliPpr Language. In *International Symposium on Haskell (Haskell)*, pages 158–171. ACM, 2018. ISBN 978-1-4503-5835-4. DOI: 10.1145/3242744.3242758. URL http://doi.acm.org/10.1145/3242744.3242758.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming (ICFP)*, pages 47–58. ACM, 2007.

M. Mayer, V. Kuncak, and R. Chugh. Bidirectional evaluation with direct manipulation. *Proc. ACM Program. Lang.*, 2 (OOPSLA):127:1–127:28, Oct. 2018. ISSN 2475-1421. DOI: 10.1145/3276497. URL http://doi.acm.org/10.1145/3276497.

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

S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *Programming Languages and Systems*, pages 2–20. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-30477-7.

D. Orchard and A. Mycroft. Categorical programming for data types with restricted parametricity. *Manuscript, submitted to Post-Proc of Trends in Functional Programming (TFP)*, 2012. https://www.cs.kent.ac.uk/people/staff/dao7/drafts/tfp-structures-orchard12.pdf.

H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 39–50. ACM, 2014a. DOI: 10.1145/2543728.2543737.

H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *International Symposium on Principles and Practice of Declarative Programming (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.

J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Programming Language Design and Implementation (PLDI)*. ACM, 2014.

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

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

T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty-printing. In *International Symposium on Haskell (Haskell)*, pages 1–12, 2010.

C. Schuster, T. Disney, and C. Flanagan. Macrofication: Refactoring by Reverse Macro Expansion. In *European Symposium on Programming (ESOP)*. Springer, 2016.

P. Stevens. Generative and transformational techniques in software engineering II. chapter A Landscape of Bidirectional Model Transformations, pages 408–424. Springer-Verlag, 2008. ISBN 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3_10.

T. van Laarhoven. CPS based functional references, Jul 2009. blog post: http://www.twanvl.nl/blog/haskell/cps-functional-references.

J. Voigtländer. Bidirectionalization for free! (pearl). In *Symposium on Principles of Programming Languages (POPL)*, pages 165–176. ACM, 2009.

P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

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.
L. Xia. Further implementations, 2018. https://github.com/Lysxia/profunctor-monad.
L. Xia, D. Orchard, and M. Wang. Composing bidirectional programs monadically (with appendices). Available at https://arxiv.org/abs/1902.06950, 2019.
L. Xia, S. Frohlich, D. Orchard, and M. Wang. Source for the paper the monadic profunctor paradigm of bidirectional programming, 2025. https://github.com/Lysxia/monadic-profunctor-bx-paper.

A Reasoning Comparison

The full definitions of the `string` parser, printer, and biparser along with backward round tripping proofs for comparison between our approach and writing two separate definitions.

A.1 Definition

Separate Parser and Printer:

```
newtype Parser a = Parser {runParser :: String → (a, String)}

instance Monad Parser where
  return x = Parser (λs → (x, s))
  p >>= k = Parser (λs →
    let (y, s') = runParser p s in
    runParser (k y) s')

parseChar :: Parser Char
parseChar = Parser (λ(c:s) → (c, s))

parseDigits :: Parser String
parseDigits = do
  d ← parseChar
  if isDigit d then do
    igits ← parseDigits
    return (d:igits)
  else if d ≡ ' ' then
    return ""
  else
    error "Expected digit or space."

parseInt :: Parser Int
parseInt = do
  ds ← parseDigits
  return (read ds)

parseString :: Parser String
parseString = do
  n ← parseInt
  replicateM n parseChar

replicateM :: Int → Parser a → Parser [a]
replicateM 0 _ = return []
```

```
replicateM n p = do
  x ← p
  xs ← replicateM (n − 1) p
  return (x:xs)

type Printer a = a → String

printString :: Printer String
printString s = show (length s) ++ " " ++ s
```

Monadic Biparser:

```
newtype BP u v = BP {unP :: (:*:)
  (Fwd (State String))
  (Bwd (WriterT String Maybe)) u v}

instance Monad (BP u) where
  return = BP ∘ return
  (BP x) >>= k = BP (x >>= (unP ∘ k))

instance Profunctor BP where
  dimap bwf fwf (BP x) = BP (dimap bwf fwf x)

instance PartialProfunctor BP where
  internaliseMaybe :: ∀u v.BP u v → BP (Maybe u) v
  internaliseMaybe (BP (Fwd py:*:Bwd qy)) = BP (Fwd py:*:Bwd handle)
    where
      -- Analogous to our previous definition of toFailure
    handle :: Maybe u → WriterT String Maybe v
    handle Nothing = WriterT Nothing
    handle (Just a) = qy a

char :: BP Char Char
char = mkBP (λ(c:s) → (c, s)) (λc → Just (c, [c]))

digits :: BP String String
digits = do
  d ← char `upon` safeHead
  if isDigit d then do
    igits ← digits `upon` safeTail
    return (d:igits)
  else if d ≡ ' ' then return " "
  else error "Expected digit or space"

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

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

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

string :: BP String String
string = int `upon` (Just ∘ length) ≫= λn → replicateBP n char

replicateBP :: Int → BP u v → BP [u] [v]
replicateBP 0 pv = return []
replicateBP n pv = do
  v ← pv `upon` safeHead
  vs ← (replicateBP (n − 1) pv) `upon` safeTail
  return (v:vs)
```

A.2 Reasoning

Separate Parser and Printer: proof that `parseString` and `printString` are backward round tripping.

Theorem A.1. For any string `s`,

```
runParser parseString (printString s) = (s, "")
```

Proof. By equational reasoning.

```
    let p = parseInt ≫= λn → replicateM n parseChar
    in runParser p (show (length s) ++ " " ++ s)
≡   runParser (replicateM (length s) parseChar) s          { Lem. A.2 }
≡   (s, "")                                                { Lem. A.4 }
```

□

Auxiliary lemmas:

Lemma A.2. For any nonnegative integer `n`, and any string `s`,

```
runParser parseInt (showInt n ++ " " ++ s) = (n, s)
```

Proof. By equational reasoning.

```
    let p = parseDigits ≫= λds → return (read ds)
    in runParser p (show n ++ " " ++ s)
≡   (read (show n), s)                                     { Lem. A.3* }
≡   (n, s)                                                 { ** }
```

* Assuming (`showInt n`) is a string of digits.

** Assuming read is a one-sided inverse of `showInt`. □

Lemma A.3. For any string of digits `ds`, and any string `s`,

```
runParser parseDigits (ds ++ " " ++ s) = (ds, s)
```

Proof. By equational reasoning, and by induction on `ds`.

- Case `ds = ""`:

```
    let arg = ("" ++ " " ++ s)
    in runParser parseDigits arg
≡   runParser parseDigits (" " ++ s)            {Def.++}
≡   runParser (parseChar >>= ...) (" " ++ s)    {Def. parseDigits}
≡   runParser (λd → if d...) s                  {Def. parseChar}
≡   runParser (return "") s                     {Def. d = ' '}
≡   ("", s)                                     {Def. runParser_return}
```

- Case `ds = (d1:ds1)`, `d1` is a digit:

```
    let arg = (d1:ds1 ++ " " ++ s)
    in runParser parseDigits arg
≡   runParser (parseChar >>= ...) arg           {Def. parseDigits}
≡   runParser (λd → if isDigit d ..) arg        {Def. parseChar}
≡   runParser (parseDigits >>= λigits →         {Def. d = d1}
    return (d1:igits)) (ds1 ++ " " ++ s))
≡   (d1:ds1, s)                                 {IH at ds1}
```

□

Lemma A.4. For any string `s`,

```
runParser (replicateM (length s) parseChar) s = (s, "")
```

Proof. By equational reasoning, and by induction on `ds`.

- Case `s = ""`:

```
    let p = replicateM (length "") parseChar
    in runParser p ""
≡   runParser (replicateM 0 parseChar) ""       {Def. length}
≡   runParser (return "") ""                    {Def. replicateM}
≡   ("", "")                                    {Def. runParser_return}
```

- Case `s = (c1:s1)`:

```
    let p = replicateM (length (c1:s1)) parseChar
    runParser p (c1:s1)
≡   runParser (parseChar >>= ...) (c1:s1)       {Def. replicateM}
≡   let p' = replicateM (length s1) parseChar   {Def. parseChar}
    >>=λs → return (c1:s)
    in runParser p' s1
≡   runParser (return (c1:s1)) ""               {IH at s1}
≡   (c1:s1, "")                                 {Def. runParser_return}
```

□

Monadic Biparser: proof that `string` is backward round tripping.

Theorem A.5. For any string `s`,

```
runParser parseString (printString s) = (s, "")
```

Proof. By compositional reasoning.
By Theorem 5.12 with Lemma A.6 and Lemma A.8 `string` is backward round tripping.

□

Auxiliary lemmas:

Lemma A.6. `string` is weak backward round tripping.

Proof. By compositional reasoning. By Proposition 5.9 with Lemma A.7. □

Lemma A.7. `char` is weak backward round tripping: for all x, y, s, s',

```
print char x = Just (y, s) ⇒ parse char (s ++ s') = (y, s')
```

Proof. By equational reasoning.
After assuming the antecedent, what remains to be proven is

```
parse char (s ++ s') = (y, s').
```

```
    parse char (s ++ s')
≡   runState (StateT (Identity ∘ (λ(c:s) → (c, s)))) (s ++ s')      { Defs. }
≡   (sh, st ++ s')                                                  { runState }
≡   (y, s')                                                         { * }
```

For the final step (*), it must be proven that `s = sh:st = y:[] = [y]`, which comes from the antecedent:

```
    print char x = Just (y, s)
≡   (runWriterT∘) (WriterT ∘ (λc → Just (c, [c]))) x =              { Defs. }
    Just (y, s)
≡   Just (x, [x]) ≡ Just (y, s)                                     { runWriterT }
```

□

Lemma A.8. For all x :: u,

```
purify string x = Just x
```

Proof. By equational reasoning.

```
    proj string xs
≡   proj (comap length int >>= λn → replicateBp n char) xs          { Defs. }
≡   (comap length (proj int) >>= λn → proj (replicateBp n char)) xs { Prop. 5.15 }
≡   (comap length Just >>= λn → proj (replicateBp n char)) xs       { Eq. (2) }
≡   proj (replicateBp (length xs) char) xs                          { Def. 4.3 }
≡   mapM (proj char) xs                                             { Eq. (3) }
≡   mapM Just xs                                                    { Eq. (1) }
≡   Just xs                                                        { Def. monad }
```

□