
Publisher's PDF, also known as Version of record
License (if available):
CC BY
Link to published version (if available):
10.1007/978-3-030-17184-1_6

Link to publication record in Explore Bristol Research
PDF-document

This is the final published version of the article (version of record). It first appeared online via Springer at https://link.springer.com/chapter/10.1007%2F978-3-030-17184-1_6 . Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research
General rights
This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available:
http://www.bristol.ac.uk/pure/about/ebr-terms
Composing Bidirectional Programs Monadically

Li-yao Xia\(^1\), Dominic Orchard\(^2\), and Meng Wang\(^3\)

\(^1\) University of Pennsylvania, Philadelphia, USA
xialiyao@seas.upenn.edu
\(^2\) University of Kent, Canterbury, UK
\(^3\) University of Bristol, Bristol, UK

Abstract. Software frequently converts data from one representation to another and vice versa. Naively specifying both conversion directions separately is error prone and introduces conceptual duplication. Instead, bidirectional programming techniques allow programs to be written which can be interpreted in both directions. However, these techniques often employ unfamiliar programming idioms via restricted, specialised combinator libraries. Instead, we introduce a framework for composing bidirectional programs monadically, enabling bidirectional programming with familiar abstractions in functional languages such as Haskell. We demonstrate the generality of our approach applied to parsers/printers, lenses, and generators/predicates. We show how to leverage compositionality and equational reasoning for the verification of round-tripping properties for such monadic bidirectional programs.

1 Introduction

A bidirectional transformation (BX) is a pair of mutually related mappings between source and target data objects. A well-known example solves the view-update problem \[2\] from relational database design. A view is a derived database table, computed from concrete source tables by a query. The problem is to map an update of the view back to a corresponding update on the source tables. This is captured by a bidirectional transformation. The bidirectional pattern is found in a broad range of applications, including parsing \[17,30\], refactoring \[31\], code generation \[21,27\], and model transformation \[32\] and XML transformation \[25\].

When programming a bidirectional transformation, one can separately construct the forwards and backwards functions. However, this approach duplicates effort, is prone to error, and causes subsequent maintenance issues. These problems can be avoided by using specialised programming languages that generate both directions from a single definition \[13,16,33\], a discipline known as bidirectional programming.

The most well-known language family for BX programming is lenses \[13\]. A lens captures transformations between sources \(S\) and views \(V\) via a pair of functions \(\text{get} : S \rightarrow V\) and \(\text{put} : V \rightarrow S \rightarrow S\). The \text{get} function extracts a view
from a source and \texttt{put} takes an updated view and a source as inputs to produce an updated source. The asymmetrical nature of \texttt{get} and \texttt{put} makes it possible for \texttt{put} to recover some of the source data that is not present in the view. In other words, \texttt{get} does not have to be injective to have a corresponding \texttt{put}.

Bidirectional transformations typically respect \textit{round-tripping} laws, capturing the extent to which the transformations preserve information between the two data representations. For example, \textit{well-behaved lenses} [5,13] should satisfy:

\[
\text{put (get } s \text{) } s = s \quad \text{get (put } v \text{) } s = v
\]

Lens languages are typically designed to enforce these properties. This focus on unconditional correctness inevitably leads to reduced practicality in programming: lens combinators are often stylised and disconnected from established programming idioms. In this paper, we instead focus on expressing bidirectional programs directly, using monads as an interface for sequential composition.

Monads are a popular pattern [35] (especially in Haskell) which combinator libraries in other 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 [17]) do not share the lens \texttt{get}/\texttt{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).

\textit{Contributions.} In this paper, we deliberately avoid the well-tried approach of specialised lens languages, instead exploring a novel point in the BX design space based on monadic programming, naturally reusing host language constructs. We revisit lenses, and two more bidirectional patterns, demonstrating how they can be subject to monadic programming. By being uncompromising about the monad 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 lenses: our interface does not rule out all non-round-tripping BXs. We tackle this issue by proposing a new compositional reasoning framework that is flexible enough to characterise a variety of round-tripping properties, and simplifies the necessary reasoning.

Specifically, we make the following contributions:

- We describe a method to enable \textit{monadic composition} for bidirectional programs (Sect. 3). Our approach is based on a construction which generates a \textit{monadic profunctor}, parameterised by two application-specific monads which are used to generate the \textit{forward} and \textit{backward} directions.

- To demonstrate the flexibility of our approach, we apply the above method to three different problem domains: parsers/printers (Sects. 3 and 4), lenses (Sect. 5), and generators/predicates for structured data (Sect. 6). While the first two are well-explored areas in the bidirectional programming literature, the third one is a completely new application domain.
We present a scalable reasoning framework, capturing notions of *compositionality* for bidirectional properties (Sect. 4). We define classes of round-tripping properties inherent to bidirectionalism, which can be verified by following simple criteria. These principles are demonstrated with our three examples. We include some proofs for illustration in the paper. The supplementary material [12] contains machine-checked Coq proofs for the main theorems. An extended version of this manuscript [36] includes additional definitions, proofs, and comparisons in its appendices.

We have implemented these ideas as Haskell libraries [12], with two wrappers around *attoparsec* for parsers and printers, and *QuickCheck* for generators and predicates, showing the viability of our approach for real programs.

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

### 1.1 Further Examples of BX

We introduced lenses briefly above. We now introduce the other two examples used in this paper: *parsers/printers* and *generators/predicates*.

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

\[
\text{parser :: String} \rightarrow \text{AST} \quad \text{printer :: AST} \rightarrow \text{String}
\]

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

\[
\text{parser} \circ \text{printer} \circ \text{parser} \equiv \text{parser} \quad \text{printer} \circ \text{parser} \circ \text{printer} \equiv \text{printer}
\]

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

For particular AST subsets, printing and parsing may actually be left- or right- inverses to each other. Other characterisations are also possible, e.g., with equivalence classes of ASTs (accounting for reassociations). Alternatively, parsers and printers may satisfy properties about the interaction of partially-parsed inputs with the printer and parser, e.g., if *parser :: String* → \((\text{AST}, \text{String})\):

\[
(\text{let } (x, s') = \text{parser } s \text{ in } \text{parser } ((\text{printer } x) \; \text{++} \; s')) \equiv \text{parser } s
\]
Thus, parsing and printing follows a pattern of inverse-like functions which does not fit the lens paradigm. The pattern resembles lenses between a source (source code) and view (ASTs), but with a compositional notion for the source and partial "gets" which consume some of the source, leaving a remainder.

Writing parsers and printers by hand is often tedious due to the redundancy implied by 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 \[17,19,30\].

**Generating and checking.** Property-based testing (e.g., QuickCheck) \[10\] expresses program properties as executable predicates. For instance, the following property checks that an insertion function \texttt{insert}, given a sorted list—as checked by the predicate \texttt{isSorted :: [Int] → Bool}—produces another sorted list. The combinator \(⇒\) represents implication for properties.

\[
\text{propInsert :: Int} \rightarrow \text{[Int]} \rightarrow \text{Property}
\]

\[
\text{propInsert val list = isSorted list \Rightarrow isSorted (insert val list)}
\]

To test this property, a testing framework generates random inputs for \texttt{val} and \texttt{list}. The implementation of \(⇒\) applied here first checks whether \texttt{list} is sorted, and if it is, checks that \texttt{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 \texttt{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 \texttt{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: \texttt{genSorted :: Gen [Int]}.

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

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

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

2 Monadic Bidirectional Programming

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

In code samples, we use the Haskell pun of naming variables after their types, e.g., a variable of some abstract 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 \( u \rightarrow m v \) (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:

```haskell
data Parser v = Parser { parse :: String -> (v, String) }
```

It is well-known that such parsers are monadic [35], i.e., they have a notion of monadic sequential composition embodied by the interface:

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

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

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

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

In practice, parsers composed with \( (>>=) \) often have a relationship between the output types of the two operands: usually that the former “contains” the latter 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 \( parser :: Parser v -> (String -> v) \).
Monadic printers. Our goal is to augment parsers with their inverse printer, such that we have a monadic type \textbf{Biparser} which provides two complementary (bi-directional) transformations:

\begin{verbatim}
parser :: Biparser v \rightarrow (String \rightarrow v)
printer :: Biparser v \rightarrow (v \rightarrow String)
\end{verbatim}

However, this type of printer \(v \rightarrow \text{String}\) (shown also in Sect. 1.1) cannot form a monad because it is \textit{contravariant} in its type parameter \(v\). Concretely, we cannot implement the bind (\(\gg\gg\)) operator for values with types of this form:

\begin{verbatim}
-- Failed attempt
bind :: ((v \rightarrow String) \rightarrow (v \rightarrow (w \rightarrow String))) \rightarrow (w \rightarrow String)
bind pv kw = \lambda w \rightarrow let v = (?) in pv v ++ kw v w
\end{verbatim}

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

But what if the type variables of \textit{bind} were related by containment, such that \(v\) is contained within \(w\) and thus we have a projection \(w \rightarrow v\)? We could use this projection to fill the hole in the failed attempt above, defining a bind-like operator:

\begin{verbatim}
bind' :: (w \rightarrow v) \rightarrow (v \rightarrow String) \rightarrow (v \rightarrow (w \rightarrow String)) \rightarrow (w \rightarrow String)
bind' from pv kw = \lambda w \rightarrow let v = from w in pv v ++ kw v w
\end{verbatim}

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

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

\begin{verbatim}
data Printer w v = Printer { print :: w \rightarrow (v, String) }
\end{verbatim}

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

\begin{verbatim}
instance Monad (Printer w) where
    return :: v \rightarrow Printer w v
    return = \lambda v \rightarrow Printer (\lambda_ \rightarrow (v, ""))

    (\gg\gg) :: Printer w v \rightarrow (v \rightarrow Printer w t) \rightarrow Printer w t
    pv \gg\gg kt = Printer (\lambda w \rightarrow let (v, s) = print pv w
                         (t, s') = print (kt v) w in (t, s ++ s'))
\end{verbatim}
The printer \texttt{return v} ignores its input and prints nothing. For \texttt{bind}, an input \texttt{w} is shared by both printers and the resulting strings are concatenated.

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

\[
\text{comap :: } (w \to w') \to \text{Printer } w' \to v \to \text{Printer } w v
\]
\[
\text{comap from } (\text{Printer } f) = \text{Printer } (f \circ \text{from})
\]

### 2.1 Monadic Biparsers

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

\[
\text{data Biparser } u \to v = \text{Biparser } \{ \text{parse :: } \text{String} \to (v, \text{String}) \\
\text{, print :: } u \to (v, \text{String}) \}
\]

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

The corresponding monad for \texttt{Biparser} is the product of the previous two monad definitions for \texttt{Parser} and \texttt{Printer}, allowing both to be composed sequentially at the same time. To avoid duplication we elide the definition here which is shown in full in Appendix A of the extended version [36].

We can also lift the previous notion of \texttt{comap} from printers to biparsers, which gives us a way to contextualize a printer:

\[
\text{comap :: } (u \to u') \to \text{Biparser } u' \to v \to \text{Biparser } u v
\]
\[
\text{comap } f (\text{Biparser } \text{parse } \text{print}) = \text{Biparser } \text{parse } (\text{print } \circ f)
\]
\[
\text{upon :: } \text{Biparser } u' \to v \to (u \to u') \to \text{Biparser } u v
\]
\[
\text{upon } = \text{flip } \text{comap}
\]

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

\textbf{An example biparser.} Let us write a biparser, \texttt{string :: Biparser String String}, for strings which are prefixed by their length and a space. For example, the following unit tests should be true:
We start by defining a primitive biparser of single characters as:

```
char :: Biparser Char Char
char = Biparser \( \lambda (c : s) \rightarrow (c, s) \) \( \lambda c \rightarrow (c, [c]) \)
```

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

Next, we define a biparser \( \text{int} \) for an integer followed by a single space. An auxiliary biparser \( \text{digits} \) (on the right) parses an integer one digit at a time into a string. Note that in Haskell, the \( \text{do} \)-notation statement "\( d \leftarrow \text{char 'upon' head} \) desugars to "\( \text{char 'upon' head} >>= \lambda d \rightarrow \ldots \)" which uses \( (\gg=) \) and a function binding \( d \) in the scope of the rest of the desugared block.

```
int :: Biparser Int Int
digits :: Biparser String String
int = do
digits = do
ds \leftarrow digits 'upon' printedInt
d \leftarrow char 'upon' head
return (read ds)
digits = do
if isDigit d then do
digits \leftarrow digits 'upon' tail
return (d : digits)
else if d == ' ' then return " ".
else error "Expected_digit_or_space"

where
printedInt n = show n ++ "_
```

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

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

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

Note that although not reflected in its type, \( \text{replicateBiparser} n pv \) expects, as a printer, a list \( 1 \) of length \( n \); if \( n == 0 \), there is nothing to print; if \( n > 0 \), \( \text{char 'upon' head} \) extracts the head of \( 1 \) to print it with \( pv \), and \( \text{char 'upon' tail} \) extracts its tail, of length \( n-1 \), to print it recursively.
Composing Bidirectional Programs Monadically

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

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

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

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

Despite its simplicity, the syntax of length-prefixed strings is notably context-sensitive. Thus the example makes crucial use of the monadic interface for bidirectional programming: a value (the length) must first be extracted to dynamically delimit the string 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.

3 A Unifying Structure: Monadic Profunctors

The biparser examples of the previous section were enabled by both the monadic structure of Biparser and the comap operation (also called upon, with flipped arguments). We describe a type as being a monadic profunctor when it has both a monadic structure and a comap operation (subject to some equations). The notion of a monadic profunctor is general, but it characterises a key class of structures for bidirectional programs, which we explain here. Furthermore, we show a construction of monadic profunctors from pairs of monads which elicits the necessary structure for monadic bidirectional programming in the style of the previous section.

Profunctors. In Sect. 2.1, biparsers were defined by a data type with two type parameters (Biparser u v) which is functorial and monadic in the second parameter and contravariantly functorial in the first parameter (provided by the comap operation). In standard terminology, a two-parameter type p which is functorial in both its type parameters is called a bifunctor. In Haskell, the term profunctor has come to mean any bifunctor which is contravariant in the first type parameter and covariant in the second.¹ This differs slightly from the standard category theory terminology where a profunctor is a bifunctor F : Dop × C →

Set. This corresponds to the Haskell community’s use of the term “profunctor” if we treat Haskell in an idealised way as the category of sets.

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

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

\begin{verbatim}
    class ForallF Functor p ⇒ Profunctor p where
        comap :: (u → Maybe u') → p u' v → p u v
    which should obey two laws:
        comap Just = id
        comap (f >>= g) = comap f o comap g
\end{verbatim}

where \((=)=)\) \((= (a → Maybe b) → (b → Maybe c) → (a → Maybe c))\) composes partial functions (left-to-right), captured by Kleisli arrows of the \texttt{Maybe} monad.

The constraint \texttt{ForallF Functor p} captures a universally quantified constraint [6]: for all types \(u\) then \(pu\) has an instance of the \texttt{Functor} class.\(^2\)

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

Since the contravariant part of the bifunctor applies to functions of type \(u → Maybe u’\), the categorical analogy here is more precisely a profunctor \(F : CT^\text{op} × C → \text{Set}\) where \(CT\) is the Kleisli category of the partiality (\texttt{Maybe}) monad.

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

\begin{verbatim}
    class (Profunctor p, ForallF Monad p) ⇒ Profmonad p
\end{verbatim}

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

\begin{verbatim}
    comap f (return y)  =  return y
    comap f (py >>= kz) = comap f py >>= (?\(λ\) y → comap f (kz y))
\end{verbatim}

\(^2\) As of GHC 8.6, the \texttt{QuantifiedConstraints} extension allows universal quantification in constraints, written as \texttt{forall u. Functor (p u)}, but for simplicity we use the constraint constructor \texttt{ForallF} from the \texttt{constraints} package: \url{http://hackage.haskell.org/package/constraints}.\n
(for all \( f :: u \to \text{Maybe} \ v \), \( p y :: p \ \text{v} \ \text{y} \), \( k z :: y \to p \ \text{v} \ \text{z} \)). These laws are equivalent to saying that \( \text{comap} \) lifts (partial) functions into monad morphisms. In Haskell, these laws are obtained \textit{for free} by parametricity [34]. This means that every contravariant functor and monad is in fact a monadic profunctor, thus the following universal instance is lawful:

\[
\text{instance } (\text{Profunctor } p, \text{ForallF Monad } p) \Rightarrow \text{Profmonad } p
\]

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

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

**Definition 3.** A value \( p :: P \ u \ \text{v} \) of a profunctor \( P \) is called \textit{aligned} if \( u = v \).

### 3.1 Constructing Monadic Profunctors

Our examples (parsers/printers, lenses, and generators/predicates) share monadic profunctors as an abstraction, making it possible to write different kinds of bidirectional transformations monadically. Underlying these definitions of monadic profunctors is a common structure, which we explain here using biparsers, and which will be replayed in Sect. 5 for lenses and Sect. 6 for bigenerators.

There are two simple ways in which a covariant functor \( m \) (resp. a monad) gives rise to a profunctor (resp. a monadic profunctor). The first is by constructing a profunctor in which the contravariant parameter is discarded, i.e., \( p \ u \ \text{v} = m \ \text{v} \); the second is as a function type from the contravariant parameter \( \text{u} \) to \( m \ \text{v} \), i.e., \( p \ u \ \text{v} = \text{u} \to m \ \text{v} \). These are standard mathematical constructions, and the latter appears in the Haskell \textbf{profunctors} package with the name \textbf{Star}.

Our core construction is based on these two ways of creating a profunctor, which we call \textbf{Fwd} and \textbf{Bwd} respectively:

\[
\begin{align*}
\text{data } \text{Fwd } m \ u \ \text{v} & = \text{Fwd } \{ \text{unFwd } :: m \ \text{v} \} \quad \text{-- ignore contrv. parameter} \\
\text{data } \text{Bwd } m \ u \ \text{v} & = \text{Bwd } \{ \text{unBwd } :: \text{u} \to m \ \text{v} \} \quad \text{-- maps from contrv. parameter}
\end{align*}
\]

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 \textbf{Fwd} and \textbf{Bwd} map any functor into a profunctor by the following type class instances:

\[
\begin{align*}
\text{instance } \text{Functor } m \Rightarrow \text{Functor } (\text{Fwd } m \ u) \quad \text{where} \\
\text{fmap } f (\text{Fwd } x) & = \text{Fwd } (\text{fmap } f \ x) \\
\text{instance } \text{Functor } m \Rightarrow \text{Profunctor } (\text{Fwd } m) \quad \text{where} \\
\text{comap } f (\text{Fwd } x) & = \text{Fwd } x \\
\text{instance } \text{Functor } m \Rightarrow \text{Functor } (\text{Bwd } m \ u) \quad \text{where} \\
\text{fmap } f (\text{Bwd } x) & = \text{Bwd } ((\text{fmap } f) \circ x) \quad \text{where} \\
\text{instance } (\text{Monad } m, \text{MonadPartial } m) \Rightarrow \text{Profunctor } (\text{Bwd } m) \quad \text{where} \\
\text{comap } f (\text{Bwd } x) & = \text{Bwd } ((\text{toFailure } \circ f) \gg> x)
\end{align*}
\]
There is an additional constraint here for $\text{Bwd}$, enforcing that the monad $m$ is a member of the $\text{MonadPartial}$ class which we define as:

\[
\text{class MonadPartial } m \ where \ \text{toFailure :: Maybe } a \rightarrow m a
\]

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

Furthermore, $\text{Fwd}$ and $\text{Bwd}$ both map any monad into a monadic profunctor:

\[
\begin{align*}
\text{instance Monad } m & \Rightarrow \text{Monad (Fwd } m u \text{) where} \\
\text{return } x & = \text{Fwd } (\text{return } x) \\
\text{Fwd } (py \gg= kz) & = \text{Fwd } (py \gg= \text{unFwd } \circ kz)
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad } m & \Rightarrow \text{Monad (Bwd } m u \text{) where} \\
\text{return } x & = \text{Bwd } (\lambda_\_ \rightarrow \text{return } x) \\
\text{Bwd } my \gg= kz & = \text{Bwd } (\lambda u \rightarrow my u \gg= (\lambda y \rightarrow \text{unBwd } (kz y) u))
\end{align*}
\]

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

\[
\begin{align*}
\text{data } (\star \star) \ p q u v & = (\star \star) \{ \text{pfst } :: p u v, \ \text{psnd } :: q u v \}
\end{align*}
\]

\[
\begin{align*}
\text{instance } (\text{Monad } p u, \text{Monad } q u) & \Rightarrow \text{Monad } ((p \star \star q) u) \ where \\
\text{return } y & = \text{return } y :\star : \text{return } y \\
py :\star : qy \gg= kz & = (py \gg= \text{pfst } \circ kz) :\star : (qy \gg= \text{psnd } \circ kz)
\end{align*}
\]

\[
\begin{align*}
\text{instance } (\text{ForallF } \text{Functor } p :\star : q, \text{Profunctor } p, \text{Profunctor } q) & \Rightarrow \text{Profunctor } (p :\star : q) \ where \\
\text{comap } f (py :\star : qy) & = \text{comap } f py :\star : \text{comap } f qy
\end{align*}
\]

### 3.2 Deriving Biparsers as Monadic Profunctor Pairs

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

\[
\begin{align*}
\text{type State } s a & = s \rightarrow (a, s) \\
\text{type WriterT } w m a & = m (a, w) \\
\text{type Biparser } & = \text{Fwd } (\text{State String} :\star : \text{Bwd } (\text{WriterT Maybe String})
\end{align*}
\]

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

\[
\begin{align*}
\text{Bwd } (\text{WriterT Maybe String}) u v & \cong u \rightarrow \text{Maybe } (v, \text{String}) \\
\text{Fwd } (\text{State String}) u v & \cong \text{String } \rightarrow (v, \text{String})
\end{align*}
\]

For the above code to work in Haskell, the $\text{State}$ and $\text{WriterT}$ types need to be defined via either a $\text{data}$ type or $\text{newtype}$ in order to allow type class instances on
Composing Bidirectional Programs Monadically

partially applied type constructors. We abuse the notation here for simplicity but define smart constructors and deconstructors for the actual implementation:

\[\text{parse} :: \text{Biparser } u \to v \to (\text{String} \to (v, \text{String}))\]
\[\text{print} :: \text{Biparser } u \to v \to (u \to \text{Maybe } (v, \text{String}))\]
\[\text{mkBP} :: (\text{String} \to (v, \text{String})) \to (u \to \text{Maybe } (v, \text{String})) \to \text{Biparser } u \to v\]

The monadic profunctor definition for biparsers now comes for free from the constructions in Sect. 3.1 along with the following instance of \text{MonadPartial} for the writer monad transformer with the \text{Maybe} monad:

\[\text{instance } \text{Monoid } w \Rightarrow \text{MonadPartial } (\text{WriterT } w \text{ Maybe}) \text{ where}\]
\[\text{toFailure } \text{Nothing} = \text{WriterT } \text{Nothing}\]
\[\text{toFailure } (\text{Just } a) = \text{WriterT } (\text{Just } (a, \text{mempty}))\]

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

The example biparsers from Sect. 2.1 can be easily redefined using the structure here. For example, the primitive biparser \text{char} becomes:

\[\text{char} :: \text{Biparser } \text{Char} \to \text{Char}\]
\[\text{char} = \text{mkBP } (\lambda (c : s) \to (c, s)) (\lambda c \to \text{Just } (c, \text{[c]}))\]

\textbf{Codec library.} The \text{codec} library [8] provides a general type for bidirectional programming isomorphic to our composite type \text{Fwd } r :*: \text{Bwd } w:

\[\text{data Codec } r \to w \to c \to a = \text{Codec } \{ \text{codecin } :: r \to a, \text{codecout } :: c \to w \to a \}\]

Though the original \text{codec} library was developed independently, its current form is a result of this work. In particular, we contributed to the package by generalising its original type (\text{codecout } :: c \to w (\cdot)) to the one above, and provided \text{Monad} and \text{Profunctor} instances to support monadic bidirectional programming with codecs.

\section{Reasoning about Bidirectionality}

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

This section studies the interaction of monadic profunctors with the \textit{round-tripping laws} that relate the two components of a bidirectional program. For every bidirectional transformation we can define dual properties: \textit{backward round tripping} (going backwards-then-forwards) and \textit{forward round tripping} (going forwards-then-backwards). In each BX domain, such properties also capture

\footnote{Smart constructors (and dually smart deconstructors) are just functions that hide boilerplate code for constructing and deconstructing data types.}
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 in Sects. 5 and 6. For brevity, we use Bp as an alias for Biparser.

**Definition 4.** A parser \( p :: \text{Bp}\ u\ u \) is **backward round tripping** if for all \( x :: u \) and \( s, s' :: \text{String} \) then (recalling that \( \text{print}\ p :: u \rightarrow \text{Maybe}\ (v, \text{String}) \)):

\[
\text{fmap}\ \text{snd}\ (\text{print}\ p\ x) = \text{Just}\ s \implies \text{parse}\ p\ (s ++ s') = (x, s').
\]

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

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

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

**Definition 5.** A parser \( p :: \text{Bp}\ u\ u \) is **forward round tripping** if for every \( x :: u \) and \( s :: \text{String} \) we have that:

\[
\text{parse}\ p\ s = (x, "") \implies \text{fmap}\ \text{snd}\ (\text{print}\ p\ x) = \text{Just}\ s
\]

**Proposition 1.** The biparser \( \text{char} :: \text{Bp}\ \text{Char}\ \text{Char} \) (Sect. 3.2) is both backward and forward round tripping. Proof by expanding definitions and algebraic reasoning.

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

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

### 4.1 Compositional Properties of Monadic Bidirectional Programming

Let us first formalize compositionality as follows. A property \( \mathcal{R} \) over a monadic profunctor \( \mathcal{P} \) is a family of subsets \( \mathcal{R}^u_v \) of \( \mathcal{P} \ u\ v \) indexed by types \( u \) and \( v \).
Definition 6. A property \( R \) over a monadic profunctor \( P \) is **compositional** if the monadic profunctor operations are closed over \( R \), i.e., the following conditions hold for all types \( u, v, w \):

1. For all \( x :: v \), \((\text{return } x) \in R^u_v\) \hspace{1cm} (comp-return)
2. For all \( p :: P u v \) and \( k :: v \rightarrow P u w \),
   \( (p \in R^{u}_{v}) \land (\forall v. (k v) \in R^{u}_{w}) \implies (p >>= k) \in R^{u}_{w} \) \hspace{1cm} (comp-bind)
3. For all \( p :: P u' v \) and \( f :: u \rightarrow \text{Maybe } u' \),
   \( p \in R^{u'}_v \implies (\text{comap } f \ p) \in R^{u}_v \) \hspace{1cm} (comp-comap)

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

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

Definition 7. A biparser \( p :: Bp u v \) is **weak backward round tripping** if for all \( x :: u, y :: v, \) and \( s, s' :: \text{String} \) then:

\[
\text{print } p \ x = \text{Just } (y, s) \implies \text{parse } p \ (s \ ++ \ s') = (y, s')
\]

Removing backward round tripping’s restriction to aligned biparsers and using the result \( y :: v \) of the printer gives us a property that is compositional:

**Proposition 2.** Weak backward round tripping of biparsers is compositional.

**Proposition 3.** The primitive biparser \text{char} is weak backward round tripping.

**Corollary 2.** Propositions 2 & 3 imply \text{string} is weak backward round tripping.

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

\[
\text{purify} :: \forall u \; v. \; \text{Bp} \; u \; v \rightarrow u \rightarrow \text{Maybe} \; v \\
\text{purify} \; p \; u = \text{fmap} \; \text{fst} \; (\text{print} \; p \; u)
\]

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

**Theorem 1.** If \(p :: \text{P} \; u \; u\) is weak backward round tripping, and for all \(x :: u\).
\[\text{purify} \; p \; x = \text{Just} \; x,\] then \(p\) is backward round tripping.

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

**Definition 8.** A *monadic profunctor homomorphism* between monadic profunctors \(P\) and \(Q\) is a polymorphic function \(\text{proj} :: P \; u \; v \rightarrow Q \; u \; v\) such that:

\[
\begin{align*}
\text{proj} \; (\text{comap}_P f \; p) &\equiv \text{comap}_Q f \; (\text{proj} \; p) \\
\text{proj} \; (p \; >>=_P k) &\equiv (\text{proj} \; p) \; >>=_Q (\lambda x \rightarrow \text{proj} \; (k \; x)) \\
\text{proj} \; (\text{return}_P x) &\equiv \text{return}_Q x
\end{align*}
\]

**Proposition 4.** The \(\text{purify} :: \text{Bp} \; u \; v \rightarrow u \rightarrow \text{Maybe} \; v\) operation for biparsers (above) is a monadic profunctor homomorphism between \(\text{Bp}\) and the monadic profunctor \(\text{PartialFun} \; u \; v = u \rightarrow \text{Maybe} \; v\).

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

**Proof** First prove (in Appendix B [36]) the following properties of biparsers \(\text{char}, \text{int},\) and \(\text{replicatedBp} :: \text{Int} \rightarrow \text{Bp} \; u \; v \rightarrow \text{Bp} \; [u] \; [v]\) (writing \(\text{proj}\) for \(\text{purify}\)):

\[
\begin{align*}
\text{proj} \; \text{char} \; n &\equiv \text{Just} \; n \\
\text{proj} \; \text{int} \; n &\equiv \text{Just} \; n \\
\text{proj} \; (\text{replicateBp} \; (\text{length} \; \text{xs}) \; p) \; \text{xs} &\equiv \text{mapM} \; (\text{proj} \; p) \; \text{xs}
\end{align*}
\]
From these and the homomorphism properties we can prove 
proj string = Just:

\[
\text{proj string } \equiv \text{proj (comap length int} \gg= \lambda n \rightarrow \text{replicateBp n char) xs}
\]
\[
\text{Prop.4} \equiv \text{comap length } (\text{proj int}) \gg= \lambda n \rightarrow \text{proj (replicateBp n char)) xs}
\]
\[
(4.2) \equiv \text{comap length Just} \gg= \lambda n \rightarrow \text{proj (replicateBp n char)) xs}
\]
\[
\text{Def.2} \equiv \text{proj (replicateBp (length xs) char) xs}
\]
\[
(4.3) \equiv \text{mapM (proj char) xs}
\]
\[
(4.1) \equiv \text{mapM Just xs}
\]
\[
\{\text{monad}\} \equiv \text{Just xs}
\]

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

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

**Definition 9.** A purifiable monadic profunctor is a monadic profunctor \( P \) with a homomorphism \( \text{proj} \) from \( P \) to the monadic profunctor of partial functions \(- \rightarrow \text{Maybe} -\). We say that \( \text{proj} \ p \) is the pure projection of \( p \).

**Definition 10.** A pure projection \( \text{proj} \ p :: u \rightarrow \text{Maybe} \ v \) is called the identity projection when \( \text{proj} \ p \ x = \text{Just} \ x \) for all \( x :: u \).

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

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

### 4.2 Quasicompositionality for Monadic Profunctors

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

Informally, an injective arrow \( k :: v \rightarrow m w \) produces an output from which the input can be recalculated:
Definition 11. Let \( m \) be a monad. A function \( k :: v \to mw \) is an injective arrow if there exists \( k' :: w \to v \) (the left arrow inverse of \( k \)) such that for all \( x :: v \):

\[
k x >>= \lambda y \to \text{return} (x, y) \equiv k x >>= \lambda y \to \text{return} (k' y, y)
\]

Next, we define quasicompositional which extends the compositionality meta-property with the requirement for \( >>= \) to be applied to injective arrows:

Definition 12. Let \( P \) be a monadic profunctor. A property \( R_{uv} \subseteq P u v \) indexed by types \( u \) and \( v \) is quasicompositional if the following holds

1. For all \( x :: v \), \( (\text{return} x) \in R_{uv} \) (qcomp-return)
2. For all \( p :: P u v, k :: v \to P u w \), if \( k \) is an injective arrow,
   \( (p \in R_{uv}) \land (\forall v. (k v) \in R_{uw}) \implies (p >>= k) \in R_{uw} \) (qcomp-bind)
3. For all \( p :: P u' v, f :: u \to \text{Maybe} u' \),
   \( p \in R_{u'v} \land \implies (\text{comap} f p) \in R_{uw} \) (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., \( s01 = s0 ++ s1 \).

Definition 13. A biparser \( p : Bp u v \) is weak forward round tripping if for all \( x :: u, y :: v, \) and \( s0, s1, s01 :: \text{String} \) then:

\[
\text{parse} p s01 = (y, s1) \land \text{print} p x = \text{Just} (y, s0) \implies s01 = s0 ++ s1
\]

Proposition 5. Weak forward round tripping is quasicompositional.

Proof. We sketch the qcomp-bind case, where \( p = (m >>= k) \) for some \( m \) and \( k \) that are weak forward roundtripping. From \( \text{parse} (m >>= k) s01 = (y, s1) \) it follows that there exists \( z, s \) such that \( \text{parse} m s01 = (z, s) \) and \( \text{parse} (k z) s = (y, s1) \). Similarly \( \text{print} (m >>= k) x = \text{Just} (y, s0) \) implies there exists \( z', s0' \) such that \( \text{print} m x = \text{Just} (z', s0') \) and \( \text{print} (k z') x = \text{Just} (y, s1') \) and \( s0 = s0' ++ s1' \). Because \( k \) is an injective arrow, we have \( z = z' \) (see appendix). We then use the assumption that \( m \) and \( k \) are weak forward roundtripping on \( m \) and on \( k a \), and deduce that \( s01 = s0' ++ s \) and \( s = s1' ++ s1 \) therefore \( s01 = s0 ++ s1 \).

Proposition 6. The \text{char} biparser is weak forward round tripping.
Corollary 4. Propositions 5 and 6 imply that string is weak forward round tripping if we restrict the parser to inputs whose digits do not contain redundant leading zeros.

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

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

Theorem 2. If \( p :: P uu \) is weak forward round-tripping, and for all \( x :: u \), \( \text{purify} \; p \; x = \text{Just} \; x \), then \( p \) is forward round tripping.

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

In summary, for any BX we can consider two round-tripping properties: forwards-then-backwards and backwards-then-forwards, called just forward and backward here respectively. Whilst combinator-based approaches can guarantee round-tripping by construction, we have made a trade-off to get greater expressivity in the monadic approach. However, we regain the ability to reason about bidirectional transformations in a manageable, scalable way if round-tripping properties are compositional. Unfortunately, due to the monadic profunctor structuring, this tends not to be the case. Instead, weakened round-tripping properties can be compositional or quasicompositional (adding injectivity). In such cases, we recover the stronger property by proving a simple property on aligned transformations: that the backwards direction faithfully reproduces its input as its output (identity projection). Appendix C in our extended manuscript [36] compares this reasoning approach to a proof of backwards round tripping for separately implemented parsers and printers (not using our combined monadic approach).

5 Monadic Bidirectional Programming for Lenses

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

```haskell
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 \( s \). This definition can be structured as a monadic profunctor via a pair of Fwd and Bwd constructions:

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

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

Ideally, get and put should be total, but this is impossible without a way to restrict the domains. In particular, there is the known problem of “duplication” [23], 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 [13] or dynamically [23]. To remedy this, we capture both partiality of get and a predicate on sources in put for additional dynamic checking. This is provided by the following Fwd and Bwd monadic profunctors:

```haskell

type ReaderT r m a = r → m a

type StateT s m a = s → m (a, s)

type WriterT w m a = m (a, w)

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

-- Smart deconstructors:

get :: L s u v → (s → Maybe v)

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

Going forwards, getting a view \( v \) from a source \( s \) may fail if there is no view for the current source. Going backwards, putting a pre-view \( u \) updates some source \( s \) (via the state transformer StateT \( s \)), but with some further structure returned, provided by WriterT \( (s → Bool) \) Maybe (similar to the writer transformer used for biparsers, Sect. 3.2). 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 \textbf{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 \textit{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. \(L \circ s \circ v \circ v\)) satisfying the standard lens well-behavedness laws, modulo explicit management of partiality via \texttt{Maybe} and testing for conflicts via the predicate:

\[
\begin{align*}
\text{put } 1 \times s &= \text{Just } ((\_, \ s'), \ p') \land p' \ s' \implies \text{get } 1 \ s' &= \text{Just } x \quad \text{(L-PutGet)} \\
\text{get } 1 \ s &= \text{Just } x \quad \implies \text{put } 1 \times s &= \text{Just } ((\_, \ s), \ _) \quad \text{(L-GetPut)}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{instance } \texttt{MonadPartial} \ (\text{StateT } s \ (\text{WriterT } (s \to \text{Bool}) \ \text{Maybe})) \ \textit{where} \\
\text{toFailure } \texttt{Nothing} &= \text{StateT } (\lambda_\_ \to \text{WriterT } \texttt{Nothing}) \\
\text{toFailure } (\texttt{Just } x) &= \text{StateT } (\lambda s \to \text{WriterT } (\text{Just } ((x , \ s), \ \texttt{mempty})))
\end{align*}
\]

\[
\begin{align*}
\text{instance } \texttt{Monoid} \ (s \to \text{Bool}) \ \textit{where} \\
\text{mempty} &= \lambda_\_ \to \text{True} \\
\text{mappend } h \ j &= \lambda s0 \to h \ s0 \ \&\& \ j \ s0
\end{align*}
\]

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

\[
\begin{align*}
\text{type } \texttt{Src} &= \text{Map } \texttt{Key } \text{Value} \\
\text{atKey : : } \texttt{Key} &\to L \texttt{Src } \texttt{Value } \texttt{Value} \quad -- \texttt{Key}-\texttt{focussed } \texttt{lens} \\
\text{atKey } k &= \text{mkLens } (\text{lookup } k) \\
&(\lambda v \to \lambda \text{map } \to \text{Just } ((v, \ \text{insert } k \ v \ \text{map}), \ \lambda m' \to \text{lookup } k \ m' \ \to \text{Just } v))
\end{align*}
\]

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

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

\[
\begin{align*}
\text{atKeys : : } \texttt{[Key]} &\to L \texttt{Src } \texttt{[Value]} \ \texttt{[Value]} \\
\text{atKeys } [] &= \text{return } [] \\
\text{atKeys } (k : \texttt{ks}) &= \text{do} \\
&\quad x \gets \text{comap } \texttt{headM } (\text{atKey } k) \quad -- \texttt{headM : : } [\texttt{a}] \to \texttt{Maybe } \texttt{a} \\
&\quad xs \gets \text{comap } \texttt{tailM } (\text{atKeys } \texttt{ks}) \quad -- \texttt{tailM : : } [\texttt{a}] \to \texttt{Maybe } [\texttt{a}] \\
&\quad \text{return } (x : \texttt{xs})
\end{align*}
\]
We refer interested readers to our implementation [12] for more examples, including further examples involving trees.

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

We first weaken backward round tripping to be compositional. Informally, the property expresses the idea, that if we put some value \( x \) in a source \( s \), resulting in a source \( s' \), then what we get from \( s' \) is \( x \). However two important changes are needed to adapt to our generalised type of lenses and to ensure compositionality. First, the value \( x \) that was put is now to be found in the output of \( \text{put} \), whereas there is no way to constrain the input of \( \text{put} \) because its type \( v \) is abstract. Second, by sequentially composing lenses such as in \( l \gg= k \), the output source \( s' \) of \( \text{put} \ 1 \) will be further modified by \( \text{put} \ 1 \ (k \ x) \), so this round-tripping property must constrain all potential modifications of \( s' \). In fact, the predicate \( p \) ensures exactly that the view \( \text{get} \ 1 \) has not changed and is still \( x \). It is not even necessary to refer to \( s' \), which is just one source for which we expect \( p \) to be True.

**Definition 14.** A lens \( 1 :: \mathcal{L} \ 	ext{s u v} \) is weak backward round tripping if for all \( x :: u, y :: v \), for all sources \( s, s' \), and for all \( p :: s \to \text{Bool} \), we have:

\[
\text{put} \ 1 \ x \ s = \text{Just} \ ((y, _), p) \land p \ s' \implies \text{get} \ l \ s' = \text{Just} \ y
\]

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

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

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

\[
\text{proj} :: s \to \mathcal{L} \ 	ext{s u v} \to (u \to \text{Maybe} \ v)
\]

\[
\text{proj} \ s = \lambda l u \to \text{fmap} \ (\text{fst} \circ \text{fst}) \ (\text{put} \ l \ u \ s)
\]

**Theorem 4.** If a lens \( 1 :: \mathcal{L} \ 	ext{s u u} \) is weak backward round tripping and has identity projections on some subset \( P \subseteq u \) (i.e., for all \( s, x \) then \( x \in P \to \text{proj} \ s \ 1 \ x = \text{Just} \ x \) then \( l \) is also backward round tripping on all \( x \in P \).

To demonstrate, we apply this result to \( \text{atKeys} :: [\text{Key}] \to \mathcal{L} \ 	ext{Src} [\text{Value}] \ [\text{Value}] \).

**Proposition 8.** The lens \( \text{atKey} \ k \) is weak backward round tripping.

**Proposition 9.** The lens \( \text{atKey} \ k \) has identity projection: \( \text{proj} \ z \ (\text{atKey} \ k) = \text{Just} \).

Our lens \( \text{atKeys} \ ks \) is therefore weak backward round tripping by construction. We now interpret/purify \( \text{atKeys} \ ks \) as a partial function, which is actually the identity function when restricted to lists of the same length as \( ks \).
Proposition 10. For all \( vs :: [\text{Value}] \) such that \( \text{length } vs = \text{length } ks \), and for all \( s :: \text{Src} \) then \( \text{proj } s (\text{atKeys } ks) vs = \text{Just } vs \).

Corollary 6. By the above results, \( \text{atKeys } ks :: \text{L } \text{Src } [\text{Value}] [\text{Value}] \) for all \( ks \) is backward round tripping on lists of length \( \text{length } ks \).

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

Definition 15. A lens \( l :: \text{L } s u v \) is \textit{weak forward round tripping} if for all \( x :: u, y :: v \), for all sources \( s, s' \), and for all \( p :: s \rightarrow \text{Bool} \), we have:

\[
\text{get } l s = \text{Just } y \land \text{put } l x s = \text{Just } ((y, s'), _) \implies s = s'
\]

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

Along with identity projection, this gives the original forward \( \text{L-GetPut} \) property.

Theorem 6 If a lens \( l \) is weak forward round tripping and has identity projections on some subset \( P \) (i.e., for all \( s, x \) then \( x \in P \Rightarrow \text{proj } s l x = \text{Just } x \)) then \( l \) is also forward round tripping on \( P \).

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

Proposition 11. For all \( ks \), the lens \( \text{atKeys } ks :: \text{L } \text{Src } [\text{Value}] [\text{Value}] \) is forward round tripping on lists of length \( \text{length } ks \).

6 Monadic Bidirectional Programming for Generators

Lastly, we capture the novel notion of \textit{bidirectional generators} (bigenerators) extending random generators in property-based testing frameworks like \textit{QuickCheck} \[10\] to a bidirectional setting. The forwards direction generates values conforming to a specification; the backwards direction checks whether values conform to a predicate. We capture the two together via our monadic profunctor pair as:

\[
\text{type } G = \text{(Fwd } \text{Gen} :*: \text{(Bwd } \text{Maybe})
\]

\[
\text{-- } \text{generate } :: G u v \rightarrow \text{Gen } v \quad \text{-- forward direction}
\]

\[
\text{check } :: G u v \rightarrow u \rightarrow \text{Maybe } v \quad \text{-- backward direction}
\]

\[
\text{mkG } :: \text{Gen } v \rightarrow (u \rightarrow \text{Maybe } v) \rightarrow G u v
\]

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

toPredicate g x = case check g x of Just _ → True; Nothing → False

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

\[ \text{instance MonadPartial Maybe where toFailure = id} \]

Due to space limitations, we refer readers to Appendix E [36] for an example of a compositionally-defined bigenerator that produces binary search trees.

**Round tripping.** A random generator can be interpreted as the set of values it may generate, while a predicate represents the set of values satisfying it. For a bigenerator g, we write \( x \in \text{generate } g \) when \( x \) is a possible output of the generator. The generator of a bigenerator g should match its predicate toPredicate g. This requirement equates to round-tripping properties: a bigenerator is sound if every value which it can generate satisfies the predicate (forward round tripping); a bigenerator is complete if every value which satisfies the predicate can be generated (backward round tripping). Completeness is often more important than soundness in testing because unsound tests can be filtered out by the predicate, but completeness determines the potential adequacy of testing.

**Definition 16.** A bigenerator \( g :: G u u \) is complete (backward round tripping) when toPredicate g x = True implies \( x \in \text{generate } g \).

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

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

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

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

\[ \text{check } g \ x = \text{Just } y \implies y \in \text{generate } g. \]

**Theorem 7.** Weak completeness is compositional.

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

**Theorem 8.** A bigenerator \( g :: G u u \) is complete if it is weak-complete and its checker satisfies a pure projection property: \( \text{check } g \ x = \text{Just } x' \implies x = x' \)

Thus to prove completeness of a bigenerator \( g :: G u u \), we first have weak-completeness by construction, and we can then show that check g is a restriction of the identity function, interpreting all bigenerators simply as partial functions.
 Considering the other direction, soundness, there is unfortunately no decomposition into a quasicompositional property and a property on pure projections. To see why, let bool be a random uniform bigenerator of booleans, then consider for example, \( \text{comap isTrue bool} \) and \( \text{comap isTrue (return True)} \), where isTrue True = Just True and isTrue False = Nothing. Both satisfy any quasicompositional property satisfied by bool, and both have the same pure projection isTrue, and yet the former is unsound—it can generate False, which is rejected by isTrue—while the latter is sound. This is not a problem in practice, as unsoundness, especially in small scale, is inconsequential in testing. But it does raise an intellectual challenge and an interesting point in the design space, where ease of reasoning has been traded for greater expressivity in the monadic approach.

7 Discussion and Related Work

Bidirectional transformations are a widely applicable technique used in many domains [11]. Among language-based solutions, the lens framework is most influential [3, 4, 13, 14, 24, 29]. Broadly speaking, combinators are used as programming constructs with which complex lenses are created by combining simpler ones. The combinators preserve round tripping, and therefore the resulting programs are correct by construction. A problem with lens languages is that they tend to be disconnected from more general programming. Lenses can only be constructed by very specialised combinators and are not subject to existing abstraction mechanisms. Our approach allows bidirectional transformations to be built using standard components of functional programming, and gives a reasoning framework for studying compositionality of round-tripping properties.

The framework of applicative lenses [18] uses a function representation of lenses to lift the point-free restriction of the combinator-based languages, and enables bidirectional programming with explicit recursion and pattern matching. Note that the use of “applicative” in applicative lenses refers to the transitional sense of programming with \( \lambda \)-abstractions and functional applications, which is not directly related to applicative functors. In a subsequent work, the authors developed a language known as HOBiT [20], which went further in featuring proper binding of variables. Despite the success in supporting \( \lambda \)-abstractions and function applications in programming bidirectional transformations, none of the languages have explored advanced patterns such as monadic programming.

The work on monadic lenses [1] investigates lenses with effects. For instance, a “put” could require additional input to resolve conflicts. Representing effects with monads helps reformulate the laws of round-tripping. In contrast, we made the type of lenses itself a monad, and showed how they can be composed monadically. Our method is applicable to monadic lenses, yielding what one might call monadic monadic lenses: monadically composable lenses with monadic effects. We conjecture that laws for monadic lenses can be adapted to this setting with similar compositionality properties, reusing our reasoning framework.

Other work leverages profunctors for bidirectionality. Notably, a Profunctor optic [26] between a source type \( s \) and a view type \( v \) is a function of type
\[ pvv \rightarrow pss, \] for an abstract profunctor \( p \). Profunctor optics and our monadic profunctors offer orthogonal composition patterns: profunctor optics can be composed “vertically” using function composition, whereas monadic profunctor composition is “horizontal” providing sequential composition. In both cases, composition in the other direction can only be obtained by breaking the abstraction.

It is folklore in the Haskell community that profunctors can be combined with applicative functors [22]. The pattern is sometimes called a monoidal profunctor. The codec library [8] mentioned in Sect. 3 prominently features two applications of this applicative programming style: binary serialisation (a form of parsing/printing) and conversion to and from JSON structures (analogous to lenses above). Opaleye [28], an EDSL of SQL queries for Postgres databases, uses an interface of monoidal profunctors to implement generic operations such as transformations between Haskell datatypes and database queries and responses.

Our framework adapts gracefully to applicative programming, a restricted form of monadic programming. By separating the input type from the output type, we can reuse the existing interface of applicative functors without modification. Besides our generalisation to monads, purification and verifying round-tripping properties via (quasi)compositionality are novel in our framework.

Rendel and Ostermann proposed an interface for programming parsers and printers together [30], but they were unable to reuse the existing structure of Functor, Applicative and Alternative classes (because of the need to handle types that are both covariant and contravariant), and had to reproduce the entire hierarchy separately. In contrast, our approach reuses the standard type class hierarchy, further extending the expressive power of bidirectional programming in Haskell. FliPpr [17,19] 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.

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) [7,9], but one loses control over the resulting distribution of generated values. Luck [15] 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.

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 monadic profunctors 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 bigen- 

erators 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 de- 

veloped two bidirectional libraries [12], 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 looka- 

head can be expressed in our extended attoparsec, but understanding its inter- 

action with (quasi)compositional round-tripping is further work.

However, this is not the final word on sequentially composable BX programs.

In all three applications, round-tripping properties are similarly split into weak 

round tripping, which is weaker than the original property but compositional, 

and purifiable, which is equationally friendly. An open question is whether an 

underlying structure can be formalised, perhaps based on an adjunction model, 

that captures bidirectionality even more concretely than monadic profunctors.

Acknowledgments. We thank the anonymous reviewers for their helpful comments.
The second author was supported partly by EPSRC grant EP/M026124/1.

References

1. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Reflections on 


of Successes That Can Change the World. LNCS, vol. 9600, pp. 1–31. Springer, 

Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_1


Database Syst. 6(4), 557–575 (1981)


lenses: alignment and view update. In: International Conference on Functional 


resourceful lenses for string data. In: Symposium on Principles of Programming 


5. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updat- 


347. ACM (2006)

6. Bottu, G.-J., Karachalias, G., Schrijvers, T., Oliveira, B.C.d.S., Wadler, P.: Quan- 


161. ACM (2017)

7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on 

Java predicates. In: International Symposium on Software Testing and Analysis 


10. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing 

of Haskell programs. In: International Conference on Functional Programming 


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.