Monadic Combinators for “Putback”
Style Bidirectional Programming

Hugo Pacheco  Zhenjiang Hu  
National Institute of Informatics, Tokyo, Japan
hpacheco@nii.ac.jp/hu@nii.ac.jp

Sebastian Fischer  
Christian-Albrechts University of Kiel, Germany
sebf@informatik.uni-kiel.de

Abstract

Bidirectional transformations, in particular lenses, are programs with a forward get transformation and a backward putback transformation that keep source and view data types synchronized. Several bidirectional programming languages exist to program in a (sort of) forward transformation, and deriving a backward transformation for free. However, the maintainability offered by such languages comes at the cost of expressiveness and (more importantly) predictability because the ambiguity of synchronization — handled by the putback transformation — is solved by default strategies over which programmers have little control.

In this paper, we argue that controlling such ambiguity is essential for bidirectional transformations and propose a novel language in which programmers write a (sort of) putback transformation, and get the unique transformation for free. Like traditional bidirectional languages, our put-oriented language allows reasoning about the correctness of defined transformations from the properties of their building blocks. But it allows programmers to describe the behavior of a bidirectional transformation much more precisely, while retaining the maintainability of writing a single program.

We demonstrate the practical power of the new approach through a series of examples, ranging from simple ones that illustrate traditional lenses to complex ones for which our putback-based approach is central to specifying nontrivial update strategies.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Algebraic approaches to semantics

Keywords  bidirectional programming, view update, lenses, Haskell, monads, generic functional programming

1. Introduction

A bidirectional transformation (BX) consists of a forward and a backward transformation that ensure the consistency of two related sources of information through modifications and evolution. During the last decade, BXs have been gaining increasing attention from a wide range of communities, including programming languages, software engineering and databases, which has motivated the proposal of a vast number of bidirectional approaches aiming to solve the problems of different bidirectional applications.

A particularly predominant BX trend within the programming languages community are functional bidirectional programming approaches, taking as main flag the pioneering work of Foster et al. on a combinatorial language for bidirectional tree transformations named lenses. Lenses recast many of the ideas from view updating in the database community. They provide forward get functions that produce a view from a source, and backward put functions that define view-update strategies describing how to “put back” modifications on a view to the source.

Definition 1.1 (Lens). A well-behaved lens \( l \) is a BX that comprises two (partial) functions \( \text{get} :: s \to v \) and \( \text{put} :: s \to v \to s \), satisfying the following properties:

\[
\begin{align*}
\text{GET} & \quad v \in \text{get} s \implies s = \text{put} s v \\
\text{PUT} & \quad s' \in \text{put} s v' \implies v' = \text{get} s'
\end{align*}
\]

A lens is said to be total if \( \text{get} \) and \( \text{put} \) are total functions.

Here \( y \in f x \) means that \( f \) is defined and that \( y = f \; x \). Note that our properties are implications and ensures that a lens is stable, i.e., whenever a view produced by \( \text{get} \) is not modified, \( \text{put} \) must return the original source. guarantees that a lens is acceptable, i.e., view updates must be translated exactly, so that if \( \text{put} \) is defined the updated view can be retrieved by applying \( \text{get} \) to the updated source.

An ad-hoc approach to bidirectional programming is to write two unidirectional transformations using standard programming languages. However, this scales badly for nontrivial transformations as we have to write and maintain two transformations. Moreover, a \( \text{get} \) function is in general not injective, so there may exist many possible \( \text{put} \) functions that combined with \( \text{get} \) form a well-behaved BX. Consider a \( \text{get} \) function that computes the height of a rectangle, written in Haskell as \( \text{height} \; (w, h) = h \). Even for this canonical example, updating the height may have different “reasonable” effects on the original width: 1) we may keep the original width; 2) we may enforce the source to be always a square with equal width and height; or 3) we may want to make the width of a predefined size if the height is modified:

\[
\begin{align*}
\text{put}_{1\text{height}} \; (w, h') & = (w, h') \\
\text{put}_{2\text{height}} \; (w, h) & = (w, h') \\
\text{put}_{3\text{height}} \; (w, h) & = (w, h') \; \text{if} \; h \equiv h' \; \text{then} \; w \; \text{else} \; i
\end{align*}
\]

This unavoidable ambiguity of \( \text{put} \) is what makes bidirectional programming challenging in practice. To ease and enable maintainable bidirectional programming, lens frameworks favor writing just a single program that can denote both transformations, and exist-
We then set to explore the inherent ambiguity of BXs farther. We attain a canonical and distinctively flexible bidirectional language whose programs can be interpreted both as a \( \text{get} \) function and a \( \text{put} \) function \( \text{get}_{\text{dual}} \) put-based combinators, what helps clarifying the (not a priori) implicit choices made by traditional lens languages that lead to their often unsatisfactory update strategies.

From the user perspective, this get-biased style simplifies bidirectional programming, but also renders it unpredictable as users have limited control over the backward direction, making it hard (or impossible) for them to specify their desired \( \text{put} \) functions using only a fixed set of strategies. For example, many existing bidirectional languages \( \text{Lens} 10 \) assume \( \text{put} \) as the most natural strategy in detriment of other possible ones. Such unpredictability hinders the adoption of BX frameworks and has been the trigger behind their boom over the past years, each proposing to answer the needs of particular bidirectional applications via different tailor-made backward semantics. \( \text{Lens} 6 \).

In this paper, we argue that the update strategy of a BX should be considered from the start, and propose a novel language of put-biased lenses entirely focused on the programming of \( \text{put} \). To change the programmer’s mindset towards writing \( \text{put} \) functions, the new combinators denote transformations oriented from view to source, in a different flavor from that of prior lens languages. We then set to explore the inherent ambiguity of BXs farther than previous work, by carefully designing each combinator with concern for not compromising expressiveness and characterizing the necessary conditions for well-behavedness and totality. As a result, we attain a canonical and distinctively flexible bidirectional language that naturally arises as a proper superset of existing lens languages: many traditional lenses can be seen as specialized versions of our (dual) put-based combinators, what helps clarifying the (not a priori clear) implicit choices made by traditional lens languages that lead to their often unsatisfactory update strategies.

For simple transformations, the effort of writing a put-based lens will be the same as writing a dual get-based lens using fixed update strategies; however, for more intricate transformations, a subtle change of combinators or default parameters will endow programmers with the necessary power to implement full-fledged update strategies. Our exercise also provides an exciting opportunity to re-evaluate where existing languages sit on this continuum.

Our main contributions can be summarized as follows:

- We propose the first attempt to carry observations about primacy of \( \text{put} \) into the design of a put-based bidirectional programming language (Section 3), and compare the combinators that arise naturally as put-oriented lenses in detail with existing lens languages.
- We enrich the structure of lenses with an abstract monadic interface to support the programming of different update strategies. By instantiating the monads, programmers can combine various classes of computational effects to elegantly refine bidirectional behavior without affecting the bidirectional properties.
- We demonstrate that put-based programming in our framework can also be programmer-friendly through several nontrivial BX examples written in our language (Section 4). Our approach allows to natively specify various update strategies that would traditionally require different tailor-made bidirectional languages.
- We describe a prototype implementation of our language as an embedded domain-specific combinator library in Haskell (Section 5) and consider possible extensions and optimizations.

Section 2 opens by identifying necessary conditions on \( \text{put} \) functions of well-behaved lenses and showing that these conditions are indeed sufficient to guarantee uniqueness of \( \text{get} \). We explain our contributions in Sections 4 and 5. Section 6 compares our approach with related work and Section 7 concludes the paper with a synthesis of the main ideas and directions for future work.

2. Put-based Bidirectional Programming

The primacy of the putback function for bidirectional (lens) programming is not a new remark and has been recognized in prior work \( \text{Lens} 6 \), in a setting where all functions are total. In this section, we generalize such results to partial well-behaved lenses and identify which of the conditions considered there arise from the well-behavedness laws and which are a consequence of added totality. We also illustrate how the combinators presented in Section 3 can be used to define a simple partial lens in putback style.

2.1 Partiality

Requiring all programs in a language to be total (as in the Agda dependently typed functional programming language \( \text{Lens} 23 \) or total bidirectional programming languages \( \text{Lens} 5,10,23 \)) requires working with precise and complex type systems and is often too restrictive in practice. Therefore, conventional functional languages such as Haskell consider programs to be partial in general.

In Haskell, a function \( f : a \to b \) can be defined by pattern matching over the input type \( a \). For example, the \( \text{tail} \) function from the standard Haskell Prelude computes the tail of a list as follows.

\[ \text{tail} (x : xs) = xs \]

Partiality can be implicitly modeled via non-exhaustive pattern matching, as for the empty list above. For non-defined patterns, we can say that \( \text{tail} [] \) is \( \bot \), being \( \bot \) a special undefined Haskell value that corresponds to the least-defined element of any type. Partiality can also be expressed via non-comprehensive guards, e.g.:

\[ \text{heightSquare} (w, h) \mid w \equiv h = h \]

This function computes the height of a square and is undefined if the input pair does not satisfy the square constraint. The domain of values on which \( f \) is defined is written \( \text{dom}(f) \). For two functions \( f \) and \( g \), inclusion \( f \subseteq g \) is defined by \( \forall a. f a \Rightarrow g a \). They are equal, \( f = g \), if \( \forall a. f a = g a \). Note that our well-behavedness laws already considered lens functions to be possibly partial. For instance, if \( \text{get} s \) is \( \bot \) for a source \( s \), then \( \text{getPut} \) does not restrict the result of \( \text{put} s v' \) for arbitrary view values \( v' \).

2.2 Put-based Laws

We now study certain implications of partial lens laws and use them to characterize partial lenses based on their \( \text{put} \) functions.

Proposition 1. For a well-behaved lens, the function “put \( s' \)” is injective for any source \( s \), in the following sense:

\[ s' \in \text{put} s v \land s' \in \text{put} s v' \Rightarrow v = v' \]

PutInj

Proposition 2. For a well-behaved lens, \( \text{putTwice} \) holds:

\[ s' \in \text{put} s v \Rightarrow s' = \text{put} s' v \]

PutTwice

While presented as necessary conditions for well-behavedness, Propositions 1 and 2 on put functions are also sufficient in the sense

\[ \text{If } \text{put} s v' \text{ is defined and equal to } s', \text{ then } \text{putGet} \text{ requires } \text{get} s' \text{ to be defined and equal to } v', \text{ however}. \]
that they give rise to a unique get such that the resulting lens is well-behaved. The main result of this section is to show that well-behaved lenses are uniquely determined by their put functions:

**Theorem 2.1** (Uniqueness of get). Assume a put function such that \( \text{putTwice} \) holds and "put s" is injective. Then the following propositions are also satisfied:

(a) For every source \( s \), there is at most one view \( v \) such that \( \text{put} s v = s \).
(b) A lens with a get function such that \( \text{get} s = v \Leftrightarrow \text{put} s v \) is well-behaved.
(c) The get function in (b) is the only one such that the resulting lens is well-behaved.

Total lenses satisfy additional surjectivity conditions, cf. [8].

### 2.3 A Taste of our Put-based Lens Language

Although writing put is semantically sufficient to uniquely determine a lens, the definition of get given in Theorem 2.1 is not efficiently implementable in a traditional programming language, as it requires finding a view \( v \) such that \( \text{put} s v = s \). In [8], we demonstrate how get functions can be derived using the Curry functional logic programming language [12]. However, complex examples require a careful implementation of put such that backtracking is available when computing get, and no assistance is provided to users in writing programs that will actually work. Moreover, backtracking is not always an efficient method to derive get.

In the next section, we propose a particular language of put-based lenses in which the get function can instead be given by construction for each lens expression. We now give a taste of our language with a simple example of a partial put-based lens.

#### Example 2.1 (List embedding).

As an example of put-based programming, consider an \( \text{embedAt i} \) function that embeds a view value at the \( i \)-th position of a source list, as illustrated in the call:

\[
\text{embedAt 2 "abcd" 'x'} = "abxd"
\]

The idea is to replace the element at position \( i \) in the source with the updated view value. In Haskell, we can write \( \text{embedAt} \) as:

\[
\begin{align*}
\text{embedAt} :: \text{Int} & \rightarrow [\text{a}] \rightarrow \text{a} \rightarrow [\text{a}] \\
\text{embedAt} 0 (s : ss) & = v : ss \\
\text{embedAt} i (s : ss) & = s : \text{embedAt} (i - 1) ss v
\end{align*}
\]

The \( \text{embedAt} \) function traverses the source list while decreasing the input index and, when the index reaches 0, replaces the head of the source with the view element. It is partial, since it is undefined for indexes larger than the length of the source list (due to the missing pattern \( \text{embedAt} i [] v \)). It is not difficult to see that this partial put function is well-behaved: \( \text{embedAt} i \) is injective for all views \( v \) whenever it is defined, and \( \text{embedAt} i (\text{embedAt} i s v) \) updates the same position in the list twice with the same view (PUTTWICE), where the second invocation returns the already updated list.

Using the put-based language introduced in Section 3 we can redefine \( \text{embedAt} \) as the following \( \text{embedAt} \) partial put-based lens.

\[
\begin{align*}
\text{embedAt} :: \text{Int} & \rightarrow ([\text{a}] \leftarrow \text{a}) \\
\text{embedAt} 0 & = \text{unhead} \\
\text{embedAt} i & = \text{untail} \circ \text{embedAt} (i - 1)
\end{align*}
\]

This higher-order function induction on the argument index to produce a lens that embeds a view value at a fixed source index.\footnote{We are using a different Sans-Serif font to differentiate lenses from the unidirectional Haskell functions such as \( \text{embedAt} \), as followed in Section 3.}

The unique get function derived by our language is Haskell’s predefined (!!) operator\footnote{The type and description of (!!) and many other Haskell standard functions used in this paper can be found via http://haskell.org/hoogle} that selects a list element by its index. Since \( \text{embedAt} \) is partial, a call like \( \text{embedAt} 2 "a" 'x' \) will fail; \( \text{embedAt} i \) is only total for source lists with length greater than \( i \). Using our language, we can systematically adapt \( \text{embedAt} \) to support this extra case, while preserving the same get, by redefining:

\[
\begin{align*}
\text{unhead'} & = \text{cons} \circ \text{keepsndOr} (\lambda x \rightarrow \text{return}[x]) \\
\text{untail'} & = \text{cons} \circ \text{keepfstOr} (\lambda y \rightarrow \text{headM} vs)
\end{align*}
\]

Here, \( \text{keepfstOr} \) is used to extend the original source list with the view value when the source list is not long enough, given \( \text{headM} [] = \emptyset \) and \( \text{headM} (x:xs) = \text{return} x; \) when the new list is already long enough, \( \text{keepnsdOr} \) returns a default empty tail. The refined \( \text{embedAt'} \) extends the source list to the necessary length by repeating the view \( \text{embedAt'} 2 "a" 'x' = "axxx" \).

### 3. A Point-free Language of Put-based Lenses

One of the reasons behind the success of get-based approaches is that users can write bidirectional programs while still thinking in a simple unidirectional way from source to view. However, \( \text{put} :: s \rightarrow v \rightarrow s \) functions are more difficult to write because users can no longer think solely in source-to-view terms, but must consider more complex update strategies that synchronize view updates with existing sources. We raise the important question: can we design a language of put functions such that more update strategies can be expressed, while offering programmability similar to writing get?

Looking at Equation 2.1, there is an essential condition independent of source values —for any source \( s \), put \( s \) (of type \( v \rightarrow s \)) must be injective— that will allow us to positively approach this question. In this section, we propose a point-free language of put-based lenses oriented from view to source, offering a set of primitive combinators described in the following core grammar.

\[
\text{Put} ::= \text{id} | \text{Put} \bowtie \text{Put} | \Phi p \bowtie \text{bot} | \text{effect} f \text{Put} | \text{in} \bowtie \text{out} | \text{addfst} f \bowtie \text{addsnd} f | \text{remfst} f \bowtie \text{remsnd} f | \text{Put} \bowtie \text{Put} | \text{inj} p \bowtie \text{Put} \bowtie \text{Put} \bowtie \text{Put}
\]

These combinators will serve as the simple building blocks to construct put-based BXs. Writing put functions in this way will assure users that corresponding lenses exist and are well-behaved: each primitive in our language is designed so that the most important premise—injectivity—is statically guaranteed by our combinators (with the exception of \( \nabla \)); and we annotate each combinator with precise typing rules that make it possible to statically prove totality based on the properties of each building block. Since different instantiations of our combinators will have identical get functions but different put update strategies, some receive extra parameters to plug in contexts where such freedom of put exists. These parameters may play different roles, and be of various types: predicates, pure functions, monadic functions, etc. The implementation of some combinators with user-provided parameters will perform local dynamic checks to guarantee that the programmer’s promise of PUTTWICE indeed holds —although debatable, we have chosen this design to really allow users to explore the inherent freedom of put, without imposing fixed default strategies. Still, in many cases such additional power is not necessary and these checks can be avoided by resorting to more usual derived combinators.

To make it more practical, we have deployed our language as an embedded domain-specific language in Haskell. Such integration enables programmers to write extra parameters in a rich familiar language without having to learn a new domain-specific language. We define our framework of put-based lenses as follows. The used
The notion of monad is captured by the following Haskell type class.

\[
\text{Monad } m \quad \text{where}
\]

- \(\text{return} :: a \to m a\)
- \(\text{bind} :: m a \to (a \to m b) \to m b\)

For particular monads, we can make this abstract notion more precise as \(x \in m \equiv (\exists h \; h \cdot m = x)\), if \(h\) is a (polymorphic) algebra for the monad at hand, essentially, a function of type \(m a \to a\) for any type \(a\). For example, for the \(\text{State}\) \(s\) \(m\) monad (presented later in Section 4) we can define \(x \in m \equiv (\exists s \cdot \text{evalState} m s = x)\).

\footnote{The \texttt{do} notation provides an alternative Haskell syntax for performing monadic computations in a more imperative programming style.}

For the context of this paper, we will assume all monadic computations (including user-provided parameters) to be totally defined, with partiality of put modeled by monadic failure. A monadic function \(f\) is said to be total if \(\forall a \cdot f\ a \neq \bot\).

In the course of this section we will describe our core put-based language, together with various common derived combinators that improve programmability in our framework and help us drawing a closer relationship with existing lens languages.

**Putlens notation** We will present each combinator in the notation shown to the right: the implementation is given in Haskell code and the Haskell type signature guarantees that (non-recursive) well-typed putlenses are well-behaved between standard Haskell types. We formulate additional semantic constraints that state the precise conditions and domains under which (non-recursive) putlenses are also total. We express these constraints in a semantic setting of sets and total functions. To highlight the difference, we refer to such semantic types using upper-case letters \(A, B, \ldots\) and total functions (and lenses) between sets of values using membership \(f \in A \to B\). Given a predicate \(p : A \to 2\) on values of type \(A\) (where the set \(2 = \{\text{True}, \text{False}\}\) corresponds to the Haskell primitive \(\text{Bool}\) type), we write \(A_p\) for \(\{a \mid a \in A \land p a\}\). We define set-theoretic unit \((1 = \{\{\}\})\), product \((A \times B = \{(x, y) \mid x \in A \land y \in B\})\) and disjoint sum \((A + B = \{\text{Left} \ x \mid x \in A\} \cup \{\text{Right} \ y \mid y \in B\})\) types.

### 3.1 Basic combinators

The identity combinator simply replaces the view for the source. View-based filtering \(\Phi\) defines subsets of the identity putlen.

\[
\begin{align*}
\text{id} & : v \equiv_{m} v \\
\Phi v & : v \equiv_{m} \Phi v \\
\end{align*}
\]

Although \(\Phi\) may seem semantically similar to \(\text{id}\), it is subtly different in that it allows users to parameterize the exact set (as a predicate or set-theoretic type) over which it is defined. We also define a special empty filter \(\bot: s \equiv_{m} v\) that is always undefined; since the empty set is a subtype of any type, its domains can be of any type.

**Sequential composition** applies a putlens \(f\) after a putlens \(g\):

\[
\begin{align*}
f & \in S \equiv_{m} U \quad g & \in U \equiv_{m} V \\
f \circ g & \equiv g \in S \equiv_{m} V \\
\end{align*}
\]

Composition first applies \(g\) (same as put \(g\)) to map the updated view to an intermediate view, and then maps the intermediate view to the source by applying \(f\), using a monadic bind. The unique get \(f\) function computes the original intermediate view of \(g\), and is applied to the “maybe” original source using \(\text{fmap} :: (a \to b) \to \text{Maybe} a \to \text{Maybe} b\) (as specialized for the Maybe type).

### 3.2 Effectful put computations

As a distinctive feature in comparison with existing bidirectional languages, our put-based language supports executing put computations augmented with an arbitrary monad, that programmers can instantiate to elegantly specify put functions incorporating different computational effects. The added monadic layer permits refining the behavior of put, but get functions remain purely functional.

\footnote{For readability, we will often omit necessary Haskell type contexts like \textit{Monad} \(m\) or \textit{Eq} \(a\) from the type signatures of putlens combinators.}
The effect combinator executes a general monadic side-effect, by running a putlens over a monad \( n \) as a putlens over a monad \( m \):

\[
\begin{array}{|c|c|}
\hline
f & \in \text{Maybe } S \rightarrow V \rightarrow n \rightarrow m \\
g & \in S \leftarrow n V \\
\hline
\text{effect} & f \; g \; \in S \leftarrow m V \\
\text{effect} & f \; g \; v' = f \; g \; v' (g \; s \; v') \\
\hline
\end{array}
\]

We write \( n \rightarrow m \) for the polymorphic function \( \forall \alpha, n \alpha \rightarrow m \alpha \). For this putlens to be well-behaved, the argument \( f \) needs to be a sort of (polymorphic) monad morphism satisfying an identity law:

\[
f \; s \; v' \; \text{return} = \text{return} \; x
\]

The kind of effects that we have in mind may not change the monad at all, like updating some internal state by increasing a counter, or run one monad and put its value inside another monad, like executing a subcomputation in a different state. Note that it does not affect well-behavedness since the bidirectional behavior of the resulting putlens is still completely determined by \( g \). This will be the case for other combinations—monadic information can only affect free choices made by put, preserving the same get.

### 3.3 Products

We also provide a set of primitive combinators to manipulate pairs. The \( \text{addfst} \) combinator (and dual \( \text{addsnd} \)) creates a pair in the source by adding a new element to the left (or right) of the view:

\[
P \subseteq S_1 \times V \quad f \in \text{Maybe } P \rightarrow V \rightarrow m \; S_1 \\
(f \; (s_1, \; v)) = \text{return} \; s_1
\]

\[
\text{addfst} \; f \in P \leftarrow m \; V
\]

\[
\text{addfst} \; f = \text{enforceGetPut} \; \text{put'} \; \text{where}
\]

\[
\text{put'} \; s \; v' = f \; s \; v' \xrightarrow{\Phi} \lambda s'_1 \rightarrow \text{return} \; (s'_1, \; v')
\]

Putting embeds the view to the second source element, while producing the first element using a user-provided function; \( f \) can be arbitrary, but it must return the original first element for the identity view update to ensure well-behavedness. In the implementation, we have two options: 1) make get undefined for sources for which \( f \) does not guarantee \( \text{PUTTWICE} \) or 2) repair the argument function by returning the original source when necessary. The latter is our only option as long as we want to enable the use of supplementary monadic information\(^6\). This is performed by the auxiliary function \( \text{enforceGetPut} \), defined later in Section 3. At the semantic level, we state a condition that user-defined functions shall statically satisfy to be well-behaved. The source domain of \( \text{addfst} \) is a dependent product \( P \), formalized as a subtype of the source product \( S_1 \times V \), because the way \( f \) constructs new source values may in general introduce a dependency between the view and source types.

The traditional projection lens \(^5\) \( [10] [24] [25] \) is the conservative variant of \( \text{addfst} \) that restores the original source if available:

\[
\forall v \in V \rightarrow m \; S_1, \; \text{keepfstOr} \; f \in S_1 \times V \leftarrow m \; V
\]

\[
\text{keepfstOr} \; (v \rightarrow m \; s_1) \rightarrow ((s_1, v) \leftarrow m v) \\
\text{keepfstOr} \; f = \text{addfst} \; (\lambda s \; v' \rightarrow \text{maybe} \; (f \; v') \rightarrow \text{return} \; \text{fat} \; s)
\]

Like many other derived combinators that we will define, \( \text{keepfstOr} \) is well-behaved by construction and does not require a dynamic check, by committing to a very particular update strategy. The typing rule becomes much simpler since \( \text{keepfstOr} \) does not introduce any dependency. A partial projection that only knows to recover the original source is given by \( \text{keepfst} = \text{keepfstOr} \; (\lambda v \rightarrow \otimes) \).

Another variant of \( \text{addfst} \) is to always ‘copy’ the updated view value in duplicate to create a new source pair:

\[
\text{copy} \; \in (V \times V) \leftarrow m \; V
\]

\[
\begin{array}{c}
\text{copy} \; v, \; v' \leftarrow m \; V \\
\text{copy} = \Phi \; \text{id} \circ \text{addfst} \; (\lambda s \rightarrow \text{return})
\end{array}
\]

The copy putlens is only total for source pairs with equal components, as modeled by the predicate \( \Phi \; (x, y) = x \equiv f \; y \). The definition explicitly restricts its source domain. The more relaxed addfst \( (\lambda s \rightarrow \text{return}) \) resembles the ‘merge’ lens from [10].

**Exercise 1 (Height).** Write the put functions from our introductory height example as putlenses using addsdn, keepsdn or copy.

Dually to addfst and addsnd, the remfst and remsnd combinators delete view pairs by discarding their left or right elements:

\[
\begin{array}{c}
\forall v \in V \rightarrow V_1, \; \text{remfst} \; f \in V \leftarrow m \; (V_1 \times V) \\
\text{remfst} \; f = \text{addfst} \; (\lambda s \rightarrow \text{return} \; f) \circ \text{remfst} \; f \subseteq \Phi \; \text{id}
\end{array}
\]

Removing an element from a pair and adding it back or adding an element and removing it afterwards are both subsets of the identity putlens. These laws use inclusion of putlenses to account for partial \( f \) functions. For a total \( f \), the laws becomes equalities; the first for the domain of pairs consistent under \( f \), and the second for any pair.

Using combinators on products, we can also define the derived ignore and new combinators that delete a concrete view or create a new source from an empty view, respectively:

\[
\begin{array}{cc}
\forall x \in V, \; \text{ignore} \; x \in 1 \leftarrow m \; x & \forall x \in S, \; \text{new} \; x \in S \leftarrow m \; 1 \\
\text{ignore} \; v \rightarrow ((1) \leftarrow m v) & \text{new} \; x \rightarrow \langle s \; \rightarrow (s \leftarrow m) \rangle
\end{array}
\]

\[
\begin{array}{c}
\text{ignore} \; v = \text{remfst} \; (\lambda () \rightarrow v) \\
\text{new} \; x = \text{remfst} \; (\lambda s \rightarrow ()) \\
\circ \circ \text{addsdn} \; (\lambda s \rightarrow \text{return} \; ()) \\
\circ \circ \text{addsdn} \; (\lambda s \rightarrow \text{return} \; x)
\end{array}
\]

The definition of ignore takes as argument a particular view \( v \) to delete, for which it is defined; new introduces a default source \( x \), but needs to return the original source if available. A combination of the two yields the ‘constant’ lens \(^5\) \( [10] [12] \).

The ‘product’ combinator applies two putlenses in parallel to distinct sides of a view pair, producing a source pair:

\[
\begin{array}{c}
f \in S_1 \leftarrow m \; V_1 \\
g \in S_2 \leftarrow m \; V_2
\end{array}
\]

\[
\begin{array}{c}
f \otimes g \in S_1 \times S_2 \leftarrow m \; V_1 \times V_2 \circ \circ (s_1 \leftarrow m \; v_1) \rightarrow (s_2 \leftarrow m \; v_2) \\
\{
\text{do:} \{ s_1' = f \; (\text{fmap \; s \; f \; s \; v')}; \\
\text{return} \; (s_1', \; s_2')
\}
\end{array}
\]

\(^6\) We could instead define a weaker version of \( \text{addfst} \) receiving a non-monadic \( f \) function and enforcing \( \text{PUTTWICE} \) by making get partial.
The monadic encoding applies $f$ to the left source/view, followed by applying $g$ to the right source/view. Depending on the monad instantiations, this chaining may induce a left-to-right evaluation order, but for simple monads (like \textit{Identity} or \textit{Reader} defined later) $f$ and $g$ can be computed in parallel. Its non-monadic variant is known as the “product” \cite{14, 23} or “concatenation” \cite{3} lens.

### 3.4 Sums

Moving to sums, the ‘injection’ combinator uses a predicate to decide whether to “tag” views as left or right values in the source:

\[
\begin{align*}
p \in \text{Maybe} \left( V_1 + V_2 \right) & \rightarrow V_1 \cup V_2 \rightarrow \text{m} 2 \\
p \left( \text{Just} \left( \text{Left} v \right) \right) & = \text{return True} \\
p \left( \text{Just} \left( \text{Right} v \right) \right) & = \text{return False}
\end{align*}
\]

\[
\text{inj} \left( \text{Maybe} \left( \text{Either} v v \right) \rightarrow v \rightarrow \text{m} \text{Bool} \right) \\
i \text{inj} p = \text{enforceGetPut put’ where put’ s v’ = p s v’} \Rightarrow \\
\lambda b \rightarrow \text{if } b \text{ then return (Left v’) else return (Right v’)}
\]

Once again, we employ \text{enforceGetPut} to dynamically enforce that put preserves the tags from the original source if the view is not modified. At the semantic level, we equate the conditions on $p$ to statically satisfy \text{PUTTWISE}. Note that the left ($V_1$) and right ($V_2$) view domains do not need to be the same and may overlap.

The specialized \text{injsOr} combinator recovers the tags from the original source if available, or uses a predicate on views instead:

\[
\begin{align*}
p & \in V \rightarrow \text{m} 2. \ \text{injsOr} \ p \in V + V \leftarrow \text{m} V \\
\text{injsOr} \ :: \ (v \rightarrow \text{m} \text{Bool}) \rightarrow \text{Either} \ v \ \text{v} \ \rightarrow \ (\text{m} \text{Left} \ v) \ \\
\text{injsOr} \ p = \text{inj} \left( \lambda s s’ \rightarrow \text{maybe} \ p s s’ \Rightarrow \left( \text{Left} \ s \right) \ s \ \left( \text{Right} \ s \right) \ s’ \right)
\end{align*}
\]

Seeing that the view is injected to the left when the original source is a left value (and vice-versa), the source domains must be the same. This embodies the behavior of the ‘either’ lens from \cite{24}.

The ‘either’ combinator (\text{\vee}) enables the specification of put lenses by case analysis and applies two different put lenses depending on the view branching, producing a source of the same type:

\[
\begin{align*}
f \in S_1 \leftarrow \text{m} V_1 \ & \ g \in S_2 \leftarrow \text{m} V_2 \ \\
f \left( \text{\vee} \right) g & \in S_1 \cup S_2 \leftarrow \text{m} V_1 + V_2 \\
\left( \text{\vee} \right) : (s \leftarrow \text{m} v_1) & \rightarrow (s \leftarrow \text{m} v_2) \rightarrow (s \leftarrow \text{m} \text{Either} v_1 v_2) \\
\left( f \left( \text{\vee} \right) g \right) s \left( \text{Left} v_1 \right) & = \text{mfilter disjoin f g} \left( f v_1 \right) \\
\left( f \left( \text{\vee} \right) g \right) s \left( \text{Right} v_2 \right) & = \text{mfilter disjoin g f} \left( g v_2 \right) \\
\text{disjoin} \ x y s & = s \in \text{dom}(\text{get} \ x) \land s \notin \text{dom}(\text{get} \ y)
\end{align*}
\]

It applies put $f$ or put $g$ to left or right view values, respectively. To guarantee \text{PUTINJ}, we must ensure the ranges of put $f$ and put $g$ (that are the domains of get $f$ and get $g$) to be disjoint – this tells us that we can later apply get $f$ or get $g$ unambiguously, thus getting a view through the same side that had put it to the source. Such test is performed by the pseudo-code of \text{disjoin}, whose particular implementation is discussed later in Section \ref{sec:composition}. Accordingly, the typing rule requires the source domains of $f$ and $g$ to be disjoint.

One way to guarantee disjointness for $\text{\vee}$ is to ask programmers to provide a predicate that declares how to branch source values:

\[
f \left( \text{\vee} p \right) g = \left( \Phi \ p \rightarrow f \right) \text{\vee} \left( \Phi \ \text{not o p} \rightarrow g \right)
\]

Here, the $\Phi$ combinator restricts the left and right source domains according to the predicate. We can also define left- ($f \text{\vee} g$) and right-biased ($f \text{\vee} g$) variants of $\text{\vee}$, that favor the domain of the left or right put lens, by instantiating $S_1 = \text{dom}(f)$ or $S_1 = \neg(\text{dom}(g))$.

The following laws reveal the duality between our injection and either combinators on sums:

\[
\left( \text{id } \text{\vee} p \right) \text{id} = \text{inj} \left( \lambda s \rightarrow \text{return o p} \right) \subseteq \text{id} \\
\Phi \left( p? \right) \text{\vee} \text{inj} \left( \lambda s \rightarrow \text{return o p} \right) = \left( \text{id } \text{\vee} p \right) \subseteq \Phi \left( p? \right)
\]

Specifically, injecting view tags with a predicate $p$ and ignoring them with $\text{\vee} p$ or ignoring view tags with $\text{\vee} p$ and re-injecting them according to $p$ are both subsets of the identity lens. If the predicate $p$ is a total function, the laws become equalities: the first for any sum and the second for sums consistent with $p$ (we define $p^\perp = \text{either} \ p \ (\text{not o p})$).

The ‘sum’ combinator (as found in \cite{14, 23}) applies two put lenses to distinct sides of the view, keeping the view branching:

\[
\left( f \text{\vee} g \right) \in s_1 \leftarrow \text{m} s_1 + s_2 \leftarrow \text{m} s_2 + s_2 \leftarrow \text{m} V_1 + V_2 \\
\Phi \left( p? \right) \text{\vee} \text{inj} \left( \lambda s \rightarrow \text{return o p} \right) = \left( \text{id } \text{\vee} p \right) \subseteq \Phi \left( p? \right)
\]

The put lenses $f$ or $g$ are applied to left/right views, producing left/right sources. The original source is considered if its branching is consistent with the view branching, or ignored otherwise.

We define standard left and right injections as derived put lenses:

\[
\begin{align*}
i \text{inj} \left( \text{Left} v \right) & = \text{return True} \\
i \text{inj} \left( \text{Right} v \right) & = \text{return False}
\end{align*}
\]

When applied to the view, \text{inj} injects the view with a left (or right) tag: \text{bot} generalizes the opposite choice to any type.

### 3.5 Recursion

In the point-free style of programming \cite{3}, algebraic data types are usually seen as sums of products. Each data type $A$ comes equipped with an isomorphism $\text{out} \in A \rightarrow FA$ that exposes its top-level structure (in a sense, encoding pattern matching over that type), and its converse $\text{inn} \in FA \rightarrow A$ that determines how values of that type can be constructed. Here, $F$ is a particular functor that represents the sums-of-products structure of type $A$. For instance, for lists we have $F A = A + A \times [A]$ and the following instances:

\[
\begin{align*}
\text{inn} \ s = \text{either} \ \left( \lambda () \rightarrow \left[ \right] \right) \left( \lambda (x, xs) \rightarrow x : xs \right) s \\
\text{out} \ l = \text{case} \ l \ of \ \left[ \right] \rightarrow \text{Left} ; (x : xs) \rightarrow \text{Right} \ (x, xs) \}
\end{align*}
\]

The \text{inn} and \text{out} isomorphisms can be lifted to the put lenses in $A \leftarrow \text{m} F A$ and out $F A \leftarrow \text{m} A$ in a trivial way. The usual constructors and deconstructors for lists are defined as follows.

\[
\begin{align*}
\text{nil} & = \text{in} \circ \text{inj} \\
\text{unnil} & = \text{id } \circ \text{bot} \\
\text{cons} & = \text{in} \circ \text{inj} \\
\text{uncons} & = \text{bot } \circ \text{id}
\end{align*}
\]

The put lenses $\text{nil}$ and $\text{unnil}$ construct/destroy an empty list, and $\text{cons}$ and $\text{uncons}$ construct/destroy a non-empty list.

Instead of introducing an explicit fixed-point combinator \cite{3}, we define recursive put lenses implicitly, relying on Haskell’s lazy recursion mechanism. Consequently, it is not guaranteed that the recursive functions of recursive put lenses terminate, and ensuring totality (and well-behavedness) \cite{5} requires tools beyond composi-
tional reasoning. The standard development to prove termination is then to introduce an ordering on putlenses and show that all descending chains of elements produced by a putlens fixed-point are finite and have a minimal element \( \lambda x.0 \). This problem has been orthogonally considered for lenses in other publications \[10,23\]. Still, we will present examples that can be proved total for particular domains.

The standard \texttt{map} function can be lifted to a (total) recursive combinator that maps an argument (total) putlens over a view list:

\[
\text{map} :: (b \ll m a) \rightarrow \left(\left[ b \right] \ll m \left[ a \right]\right) \\
\text{map} f = \text{in} \circ (\text{id} \otimes f \circ \text{map} f) \circ \text{out}
\]

We can also define a putlens for the ubiquitous \texttt{foldr} function:

\[
\text{unfoldr} :: \left(\left( b, a \right) \ll m a \right) \rightarrow a \rightarrow \left(\left[ b \right] \ll m \left[ a \right]\right) \\
\text{unfoldr} f x = \text{in} \circ \text{iterate} \circ (\text{inj} \circ \text{return} \circ (\equiv x)) \\
\text{where} \text{iterate} = \text{ignore} \circ (\text{id} \otimes \text{unfoldr} f x) \circ f
\]

The \texttt{unfoldr} putlens\(^9\) takes a putlens \( f \) and a default view value \( x \). Its get function will perform just as the (uncurried) \texttt{foldr} function, and fold a list of type \( \left[ b \right] \) in a bottom-up approach to produce a value of type \( a \), by applying \( f \) at each iteration and returning the default \( x \) for the empty list. Its put function will unfold the view of type \( a \) to produce an updated source list (while consuming the original source list in some way). An interesting detail is how \texttt{iterate} tests if the view matches the default \( x \) to decide whether to stop recursion in the backward direction (generating the \([\ ]\) list) or not. For this reason, \texttt{unfoldr} will only build a total well-behaved putlens if the given \( f \) somehow converges into a minimal element \( x \).

### 3.6 Injective functions are putlenses

The simplest cases of BXs are isomorphisms. Given an injective function \( f \) and its inverse \( f^{-1} \), we can trivially build a lens with \texttt{put} \( s = f \) and \texttt{get} \( f^{-1} \). In existing combinatorial BX languages \[21,22\], the variable-free style used to specify BXs requires making the control flow explicit through the use of some “piping” isomorphisms. These combinatorial BX languages allow an important role in extending the expressiveness of the bidirectional language (as lens categories \[21,22\] do not support categorical sums and products). For example, the following piping combinators reflect the commutativity, associativity and distributivity of sums and products:

\[
\text{swap} :: (b, a) \ll m (a, b) \\
\text{assocl} :: (a, (b, c)) \ll m (a, \left( b, c \right)) \\
\text{coswap} :: \text{Either } a \otimes \text{Either } a \otimes b \\
\text{coassocl} :: \text{Either } \left( \text{Either } a \otimes b \right) c \ll m \text{Either } a \otimes \text{Either } b \otimes c \\
\text{distl} :: (a, (b, c)) \ll m \text{Either } a \otimes \text{Either } b \otimes c \\
\text{undistl} :: \text{Either } a \otimes \text{Either } b \otimes c \ll m \left( a, \left( b, c \right) \right)
\]

The putlens inverses for these combinators are correspondingly named \texttt{assocr}, \texttt{coassocr}, \texttt{distr} and \texttt{undistr}. We can define all these combinators not as primitives but as derived (partial) putlenses, by lifting their standard unidirectional definitions to putlenses.

### 3.7 Conditionals are putlenses

Using putlenses on sums, we can define a conditional combinator:

\[
\text{ifthenelse } f g p f g \left( f \bigotimes g \right) \circ \text{inj } p
\]

It takes two putlenses \( f \) and \( g \) and a predicate \( p \) deciding to apply either \( f \) or \( g \). When the source domains of the two putlenses overlap, \( f \) is preferred. Despite general, it is not very interesting by itself as its semantic constraints and behavior just combine those of \( \text{inj} \) and \( \bigotimes \): source domains must be disjoint and view domains can overlap.

To avoid overlapping between the source domains of \( f \) and \( g \), we can define specific conditionals that work over more constrained domains. Consider the view-based ‘if-then-else’ combinator:

\[
\begin{align*}
V_1 \subseteq V & \quad f \in S_1 \leftarrow m V_1 \quad g \in S \setminus S_1 \leftarrow m V \setminus V_1 \\
\text{ifVthenelse } V_1 f g \in S \leftarrow m V
\end{align*}
\]

\[
\begin{align*}
\text{ifVthenelse } (v \mapsto \text{Bool}) & \rightarrow (s \leftarrow m v) \rightarrow (s \leftarrow m v) \\
& \rightarrow (s \leftarrow m v) \\
\text{ifVthenelse } p f g & = (\left( f \mapsto \Phi \right) \bigotimes g) \circ \text{inj } (\lambda s \mapsto \text{return } \circ p)
\end{align*}
\]

Given a user-specified predicate \( p \) on views, it applies either \( f \) or \( g \) depending on \( p \). At the semantic level, \( f \) and \( g \) must map disjoint view domains (consistent with \( p \)) to disjoint source domains, thus behaving as some sort of ‘sum’ combinator on sets instead of disjoint sums. It is equivalent to the \texttt{acond} combinator from \[10\].

We can also provide a source-based ‘if-then-else’ combinator:

\[
\begin{align*}
S_1 \subseteq S & \quad f \in S_1 \leftarrow m V \quad g \in S \setminus S_1 \leftarrow m V \\
\text{ifSthenelse } S_1 f g \in S \leftarrow m V
\end{align*}
\]

\[
\begin{align*}
\text{ifSthenelse } (s \mapsto \text{Bool}) & \rightarrow (s \leftarrow m v) \rightarrow (s \leftarrow m v) \\
& \rightarrow (s \leftarrow m v) \\
\text{ifSthenelse } p f g & = (f \nabla p) g \circ \text{inj } \text{Or } (\lambda v \mapsto \text{return } \text{True})
\end{align*}
\]

This time, the branching is decided by a predicate \( p \) on sources applied to the original source (if it is unavailable, then \( f \) is preferred). As a consequence of using \texttt{inj}Or, the view domains of \( f \) and \( g \) must be identical. This corresponds to the \texttt{acond} combinator from \[10\].

Conditionals also allow to encode pattern matching without mentioning in and out, in favor of more intuitive constructors or destructors, as in the following alternative formulation of \texttt{map}:

\[
\text{map } f = \text{ifVthenelse } \text{null } (\text{nil } \otimes \text{unnil}) \text{ iterate} \quad \text{where} \text{ iterate } = \text{cons } \otimes (f \otimes \text{map } f) \otimes \text{uncons}
\]

### 3.8 User-defined putlenses

To make our Haskell library of putlenses extensible, we also admit user-defined lenses. These are useful in practice to define putlenses for primitive types (like integers or strings) directly in terms of their primitive operations. Any custom lens (a pair of well-behaved \texttt{put} and \texttt{get} functions) can be encoded using custom:

\[
\text{custom } (\text{\begin{align*}
\text{Map} & \rightarrow v \mapsto v \mapsto m s & \rightarrow (s \mapsto v) & \rightarrow (s \leftarrow m v) \\
\text{custom } \text{put} & = \text{remfst} & \text{get } & \circ \text{addsnd} & (\lambda p \mapsto \text{put } (\text{fmap snd } p))
\end{align*}})
\]

Custom putlenses are always well-behaved\[^10\] but should only be used for simple transformations since programmers must write two \texttt{put} and \texttt{get} functions by hand, without bidirectional support.

### 4. Put-based Lens Programming

In this section, we argue for specifying a well-behaved BX by writing a put-based lens in our language. We illustrate the spirit of put-based programming through several examples that highlight the features of our language, and discuss how their corresponding put functions can be programmed and instantiated with different monadic effects to reflect different update strategies.

#### 4.1 Common monadic effects

Thus far, the presentation of our language has only assumed an abstract monadic interface, not imposing any special structure on computational effects. Naturally, concrete monads are likely to be used and combined for particular examples.

\[^9\] We use the name \texttt{unfoldr} to denote the view-to-source putlens for the \texttt{foldr} forward function and not a lens version of the Haskell \texttt{unfoldr} function.

\[^{10}\] The custom combinator ensures well-behavedness of the resulting putlens by possibly making it less defined than the passed functions.
The `Identity` monad models pure function application, and comprises a constructor `Identity a`, with a trivial algebra `runIdentity :: Identity a -> a` that simply retrieves a pure value. We can program in a language of traditionally pure putlenses by instantiating our combinators with the `Identity` monad. For putlenses without monadic effects, we often write `s <=< v` as short for `s <=< Identity v`.

The `Reader r` monad models computations that access a shared environment `r`. We can run a reader-aware putlens `l :: s <=< Reader r v` in putback direction using an algebra `runReader (l s v') r`, for some initial environment `r`, source `s` and view `v'`. We can easily lift specific monadic operations to putlenses using effect, often write `runReader` as: for three possible `Identity a`

The intention is that `runReader` combinators runs a putlens computation in a newly initialized environment. For the sake of generalization, the `Reader r` monad transformer (`Reader r = ReaderT r Identity`) adds a reader environment to any other monad.

Perhaps more familiar is the `State s a` monad of computations using an internal state `s`. Additionally to reading from a shared state (`runState :: State s a -> s`, computations may also modify the current state via an operation `writeState :: s -> State s a`). We can define similar lifted combinators over states, like `runState` that runs a putlens computations under an initialized state:

```
runState :: (Maybe s -> v -> m st) 
    -> (s <=< StateT st v m) -> (s <=< m v)
```

A particular subclass of monads that permit computations to recover from failure is captured by the `exception` monad:

```
class Monad m => MonadExcept m where 
    catch :: m a -> m a 
```

The intention is that `catch` raises exceptions and `catch` `m h` can capture failures of `m` and pass control to the handler `h`, satisfying sensible laws. A typical instance of `MonadExcept` is the `Maybe` monad, with an algebra from `Just :: Maybe a -> a`.

Despite these are the monads that we found most useful for the coming examples, other interesting monads could be considered. For instance, a list or probabilistic monad could be useful for specifying update strategies in a functional logic programming style or for lenses that are non-deterministic in general.

4.2 Defining putlenses using structural recursion

As common in functional programming, our language allows writing putlenses using structural recursion patterns that hide recursion from the users, like map and unfold from Section 4.1. More importantly, we can express different update strategies in this style.

**Example 4.1** (Update sum of list). Imagine a `put` function that, given a source list of numbers and an updated view number, modifies the source list so that its summation becomes the updated view:

```
summends :: [Int] -> Int -> [Int]
```

There are naturally many ways to update a list so that it meets a particular sum. For instance, consider the following `put` functions for three possible `summends` putlenses.

```
> get [1,2,3,4] 10 > putl1 [1,2,3,4] 15 [1,2,3,4,5]
> putl2 [1,2,3,4] 15 > putl3 [1,2,3,4] 15 [2,3,4,6]
```

The `get` function is the same for the three putlenses: the Haskell `sum` function that sums a list of numbers. Our first strategy (`putl1`) preserves the original source and appends the view update (the modified view subtracted by the original view) as a last element of the updated source. Our second strategy (`putl2`) recursively traverses the source list and divides the view update by two at each recursive step: half of the view update is added to the first element of the source, a quarter to the second, and so on until the remainder is 0. Our third strategy (`putl3`) behaves similarly to the second one but instead divides the view update by the length of the source, to distribute the difference evenly among the original source elements.

In order to encode these strategies, we start by hand-coding an arithmetic putlenses that splits a view integer into two summands:

```
split :: (Int -> Int -> m Int) -> ((Int, Int) <=< m Int) 
split offset = custom put (uncurry (+)) where 
    put (Just (x, y)) z = do i <- offset (x + y) z return (x + i, z - x - i)
    put Nothing z = return (z, 0)
```

The corresponding get function is just (uncurried) binary addition. To support different splitting behaviors, the auxiliary function `offset` computes the offset to be added to the original left value, while the remainder of the view modification is added to the original right value. The resulting putlens is total for any total offset argument function. We can now encode the three strategies (`putl1`, `putl2`, `putl3`) for `summends` using a recursive unfoldr and different offsets:

```
summends1, summends2, summends3 :: [Int] <=< m Int 
summends1 = unfoldr (split (λ (x, y) → return (x + y))) 0 
summends2 = unfoldr (split (λ (x, y) → return (x + y))) 2) 0 
summends3 = runState (λ (x, y) → return (x + y)) (unfoldr split 0)
```

The put of `summends2` keeps the original source and appends the view update as a last element, because its underlying split always adds 0 to the original left value. For `summends3`, split divides the view update in half. To evenly split the view update, `summends2` needs more information at the time it (binarily) splits a view integer; we extend it with an additional state that counts the length of the original source, decreased at each put iteration. These three variants can be proved total, by observing that split will return a pair `(z, 0)` when the original source list is empty, what will produce a singleton list `[z]` according to the stop condition of `unfoldr`.

4.3 Defining putlenses in analogy to get-based style

Users can also program putlenses by “reversing” existing unidirectional programs, written from source to view, in a style dual to traditional get-based bidirectional languages.

**Example 4.2** (Insertion “unsort”). A simple sorting algorithm is insertion sort, that can be encoded in Haskell as follows.

```
insort :: Ord a => [a] -> a ins :: Ord a => a -> [a] -> [a]
insort [] = [] ins [] = x
insort (x:ss) = ins (y:ys) = if x <= y then x : (isort xs)
```

Consider now that we want to write a putlens that “unsorts” a list by reversing `isort`. It may be useful, for example, to compose with other lenses that assume sorted lists. Since we consider view lists to be always sorted, a put function for `isort` has the liberty to “disarrange” the elements in the view as long as it preserves the same elements. We consider two of many possible update strategies:

```
12 Note that custom will enforce `PUTTWICE` for any offset function.
11 Remember that for, runReader to be well-behaved, programmers must ensure that f (Just s) (get s) = return r, for some environment r.
```
Our first strategy (iunsort₁) just disarranges the view based on the "disarrangedness" of the source, i.e., the view list is "unsorted" in reverse way to which the original source was sorted (up to the length of the original source). But other strategies may freely disarrange the view in a different way, like iunsort₂ that simply copies the (sorted) view to the source whenever the view is modified.

We can define the first putlens by dualizing isort to iunsort₁ and ins to del using default conditional combinators as follows.

\[
\begin{align*}
\text{iunsort₁} &:: \text{Ord a} \Rightarrow ([a] \imies [a]) \\
\text{iunsort₁} &\equiv \text{ifThenElse} \text{null} \text{null} \text{nil} \text{nil} \text{null} \\
\text{where} &\quad \text{iterate} = \text{cons} \circ (\text{id} \otimes \text{iunsort₁}) \otimes \text{del} \\
\text{del} &\equiv \text{ifThenElse} \text{null} \otimes \text{snd} \text{id} \text{disarrange} \circ \text{uncons} \\
\text{where} &\quad \text{disarrange} = \text{ifThenElse} \text{p} \text{id} \text{reorder} \\
&\quad \text{p} (x, y : ys) = x \leq y \\
&\quad \text{reorder} = (\text{id} \otimes \text{cons}) \circ \text{subr} \circ (\text{id} \otimes \text{del})
\end{align*}
\]

The piping combinator subr (that swaps the order of \(x\) and \(y\) in the code of ins) is defined as associ \(\circ (\text{swap} \otimes \text{id}) \circ\text{assocl}\). The definition of iunsort₁ essentially consists in removing variables, and the most interesting part lies in the code of disarrange, that decides if the order of the view shall be preserved in the source (id) or if view elements shall be reordered in some way (reorder). The default behavior of iunThenElse will preserve the relative order of the elements in the source. Other strategies can be simulated by adequately changing the code of disarrange, e.g., iunsort₂ is defined using disarrange = iunThenElse mod (\(\Phi p\)) reorder, with mod (Just (Right v)) \(v = \text{return}\ False\) and \(\text{mod}\ s\ v = \text{return}\ True\) otherwise. As expected, both iunsort₁ and iunsort₂ are total for all source lists and sorted view lists.

### 4.4 Defining flexible alignment strategies

For state-based BXs (in contrast to operation-based BXs \(\text{[4]}\)), the put function of a lens must align the original source with the modified view to identify view modifications and translate them to the source. This alignment problem is well-known in the literature, and some languages \(\text{[2]}\ \text{[5]}\ \text{[25]}\) promote decomposing update strategies into: an explicit alignment phase, that infers an high-level description of a view update between source and view structures; and a backward transformation phase, that makes use of the inferred information to guide the propagation of view modifications to the source. Our language can flexibly specify various alignment strategies, as we now illustrate with the encoding of a typical database projection operation.

For the sake of simplicity, we regard database tables as sets of rows represented as lists sorted by a key that identifies rows uniquely. In Haskell, a row can be seen as a record type with the names of columns as fields. Consider a record of people and a conforming database where each person is identified by its name:

\[
\begin{align*}
\text{data} \quad \text{Person} &= \text{Person} \{ \text{name} := \text{Name}, \text{city} := \text{City} \} \\
\text{type} \quad \text{Name} &= \text{String} \\
\text{type} \quad \text{City} &= \text{String} \\
\text{people} &= [\text{peter}, \text{roberto}, \text{david}] \\
\text{peter} &= \text{Person} \ "\text{Peter}" \ "\text{San Diego}\" \\
\text{roberto} &= \text{Person} \ "\text{Roberto}" \ "\text{Rome}\" \\
\text{david} &= \text{Person} \ "\text{David}" \ "\text{San Diego}\"
\end{align*}
\]

The following putlenses allow modifying the fields of people.

\[
\begin{align*}
\text{name} :: \text{Person} \leftarrow \text{Name} &\quad \text{city} :: \text{Person} \leftarrow \text{City} \\
\text{name} = \text{in} \otimes \text{keepsnd} &\quad \text{city} = \text{in} \otimes \text{keepfst}
\end{align*}
\]

Example 4.3 (DB projection). Using a bidirectional language, we can easily define a lens like the one below that projects only names from of a database of people, ignoring their cities.

\[
\begin{align*}
\text{peopleNames} : \text{City} \rightarrow [\text{Person}] \leftarrow m [\text{Name}] &\quad \text{peopleNames} \ \text{newc} = \text{map} \ (\text{in} \otimes \text{addsnd} \ \text{cityOf}) \\
\text{where} &\quad \text{cityOf} s\ v = \text{return} \ (\text{maybe} \ \text{newc} \ \text{snd} \ s)
\end{align*}
\]

This projection putlens maps addsnd cityOf over a list of view names, applying cityOf to retrieve the city of each person’s name if a city exists in the original database, or creating a new city otherwise. However, the behavior of this putlens is highly unsatisfactory. Consider the following invocation:

\[
\begin{align*}
&\text{> let} \ s = [\text{roberto, david}] \\
&\text{> let} \ v' = [\text{"Peter", "Roberto", "David"}] \\
&\text{> runIdentity} \ (\text{put} \ \text{peopleNames} \ (\text{"Lyon"}) \ s\ v') \\
&[\text{Person} \ "\text{Peter}" \ "\text{Rome}\", \text{Person} \ "\text{Roberto}" \ "\text{San Diego}\", \text{Person} \ "\text{David}" \ "\text{Lyon}\"
\end{align*}
\]

By adding Peter to a list of names with only Roberto and David, this naive put will align people in the two lists positionally instead of by name, what results in the incorrectly matched cities. This problem occurs because, as in a traditional bidirectional language, map simultaneously traverses source and view lists, and addsnd can only see the old source element for a view element at the same position, but not the whole source list. To align people by their names, we can introduce an environment that remembers the associations between names and cities in the original source, and extend cityOf to reuse such information (provided by the monad):

\[
\begin{align*}
\text{peopleNames} : \text{City} \rightarrow [\text{Person}] \leftarrow \text{Reader} [\text{Person}] \ [\text{Name}] &\quad \text{peopleNames} \ \text{newc} = \text{map} \ (\text{in} \otimes \text{addsnd} \ \text{cityOf}) \ \text{where} \\
&\quad \text{cityOf} s\ n = \text{ask} \ \text{read}\ (\text{people} p \rightarrow \text{return}) \\
&\quad \text{(case} \ \text{find} \ (\lambda p \rightarrow \text{get name} \ p \equiv n) \ \text{people} \ (\text{Just} \ p \rightarrow \text{get city} \ p; \text{Nothing} \ → \text{newc})
\end{align*}
\]

Initializing the reader with the associations from the original source, newly inserted names are now processed accordingly:

\[
\begin{align*}
&\text{> runReader} \ (\text{put} \ \text{peopleNames} \ (\text{"Lyon"}) \ s\ v') \ s \\
&\quad [\text{Person} \ "\text{Peter}" \ "\text{Lyon}\", \text{roberto, david}]
\end{align*}
\]

Although we have just demonstrated a simple key-based example, virtually any alignment-based strategy can be simulated using a Reader monad: the original source and modified view are pre-aligned as a separate aspect controlled by users, and the inferred associations initialize an environment that is pipelined through put transformations. BX developers could then devise alignment-aware variants of putlens combinators that use and propagate such alignment information, hiding the internal plumbing from the users in the same way other alignment-aware languages \(\text{[2]} \text{[5]} \text{[25]}\) do.

### 4.5 Defining various database view-update strategies

Historically, database view updating is a primary source of motivation for research in BXs. Apart from very restrictive scenarios, view updating is inherently ambiguous and some existing solutions propose allowing users to control the update strategy to some extent. We now demonstrate how similar user-parameterizable database views can be encoded in our put-based language.

Example 4.4 (DB selection). One typical operation for defining views of databases is selection, which filters rows satisfying a given condition. Reflecting view insertions and modifications is straightforward: we must copy such rows to the source. When source rows are deleted (either by selection or by the update), however, we may either reasonably: 1) delete them from the original database table, or 2) change them so that they do not satisfy the selection
condition. Keller [16] uses this as a classical example to illustrate the ambiguity of view update translation.

Consider a database query peopleFrom that selects people from a city from, and two update strategies: peopleFrom that removes deleted people in the view, and peopleFromTo to that moves deleted people to a new city to different than from:

\[ \text{select} :: \text{Ord } k \Rightarrow ([a] \rightarrow ([a] \leftarrow m [a])) \rightarrow ([a] \leftarrow m [a]) \]

\[ \text{select} \text{ key entries } p \rightarrow \text{runState } \text{rs} \rightarrow \text{select' key } p \] where

\[ \text{rs} \rightarrow \text{entries } s \Rightarrow \lambda s \rightarrow \text{return } (\text{Nothing}, \text{rs}) \]

\[ \text{select' key } p \rightarrow \text{ifthenelse cond recover iterate where} \]

\[ \text{cond } s \rightarrow v \leftarrow \text{do } (\_ \rightarrow \text{readState } \text{let } (h, t) = \text{recoverRow key } v' \rightarrow \text{writeState } (h, t) \rightarrow \text{return } (\text{isJust } h) \]

\[ \text{recover } = \text{cons } \rightarrow (\Phi (\text{not } o p) \otimes \text{select' key } p) \rightarrow \text{addfst} \]

\[ \\lambda v \rightarrow \text{do } \{(x, x) \leftarrow \text{readState}; \text{return } x \} \]

\[ \text{iterate } = \text{in } \rightarrow (\text{id } \oplus \text{id } \otimes \text{select' key } p) \rightarrow \text{out} \]

\[ \text{recoverRow key } [] \rightarrow (\text{Nothing}, []) \]

\[ \text{recoverRow key } (x : xs) \leftarrow (\text{Just } x, xs) \]

\[ \text{recoverRow key } (x : xs) \leftarrow (v : vs) \]

\[ \text{recoverRow key } (x : xs) \leftarrow (x : xs) \]

\[ \text{Our select combinator takes as arguments a predicate } p \text{ (the filtering criterion), a } \text{key} \text{ function that uniquely determines rows, and an } \text{entries} \text{ function that computes a subtable of rows (not satisfying } p \text{) to be merged with the updated view. It starts by initializing a state monad with such subtable; then, it either recovers a row from the state (recover) or proceeds recursively by copying a row from the view (iterate). The auxiliary function recoverRow controls this choice: it takes two (we assume sorted) lists of "recoverable" rows and view rows, and recovers a row if it has key smaller than the first view row (or there are no view rows), updating the state.} \]

We define peopleFromTo by initializing the state of select with only the people not in from, and peopleFromToTo by considering also the people originally in from but moved to city to:

\[ \text{peopleFromTo :: City } \rightarrow ([\text{Person} ] \leftarrow m [[\text{Person} ]]) \]

\[ \text{peopleFromTo from select get name } rs \leftarrow \text{(isFrom from)} \]

\[ \text{where } rs = \text{return } o \text{ maybe } (\{ \text{filter } (\text{not } o \text{ isFrom from)} \}) \]

\[ \text{peopleFromTo from to select get name } rs \leftarrow \text{(isFrom from)} \]

\[ \text{where } rs = \text{return } o \text{ maybe } (\{ \text{map move } p \leftarrow \text{if isFrom from } p \text{ then put city } p \rightarrow \text{else } p \}

\[ \text{Both putlenses are total for view lists satisfying } p, \text{ and peopleFromToTo only for cities to } \neq \text{ from. (Lists don’t need to be especially sorted – we just assumed so to clarify our exposition.) Here, we define the predicate } \text{isFrom from } p = \text{get city } p \equiv c. \]

Using our bidirectional semantics for selection, we can also define a relational join putlens. Strategies for joins are more complex because there is more ambiguity on which source information to delete when a row is deleted from the updated join. We refrain from presenting this combinator due to space limitations.

4.6 Defining putlenses by program inversion

In our language, put functions may innately fail for partial putlenses. Imagine that we would like to combine two putlenses unnil and uncons into a single putlens that processes both empty and non-empty lists. Being so, we need to account for the event that unnil may fail for non-empty views, and apply uncons instead. Using \textit{catch}, we can express this behavior as a ‘union’ putlens:

\[ f \in S_1 \Leftarrow m V_1 \]

\[ g \in S \setminus S_1 \Leftarrow m V_2 \]

\[ \text{union } f \rightarrow g \in S \Leftarrow m V_1 \cup V_2 \]

\[ \text{union } \phi \rightarrow g \rightarrow (\Phi (\text{dom } (g) ) ) \rightarrow \text{id } \rightarrow \text{inju where} \]

\[ \text{inju Nothing } v = (f \rightarrow \text{Nothing } v \triangleleft \text{return } o \text{ Left}) \]

\[ \text{catch } (g \rightarrow \text{Nothing } v \triangleleft \text{return } o \text{ Right}) \]

\[ \text{inju (Just } s) v = (f \rightarrow \text{Just } s \rightarrow \text{return } o \text{ Left}) \]

\[ \text{catch } (g \rightarrow \text{Nothing } v \triangleleft \text{return } o \text{ Right}) \]

\[ \text{inju (Just } s) v = (f \rightarrow \text{Just } s \rightarrow \text{return } o \text{ Right}) \]

\[ \text{catch } (f \rightarrow \text{Nothing } v \triangleleft \text{return } o \text{ Left}) \]

The typing rule for this combinator is similar to that of our general conditional combinator, allowing the view domains to overlap but forcing the source domains to be disjoint. (As an exercise, readers are invited to write this putlens as a derived form of ifthenelse, but in a less effective way.) The putback direction applies \textit{f} or \textit{g}, preferring \textit{f} when the view domains overlap except if the original source belongs to the source domain of \textit{g}. It can be found in the literature as the \textit{cond} [10] or ‘union’ [5] lens. The ‘union’ lens of \textit{f} follows the same spirit of our auxiliary inju combinator. A similar form of shallow backtracking is used for the ‘junc’ combinator of [29].

Example 4.5 (Tokenize words). The Haskell \textit{unwords} function joins a list of words with a separating space. Using union, we can write an inverse putlens that parses a view string into a list of words:

\[ \text{words } \cdot\cdot\cdot \text{MonadExcept } m \Rightarrow [\text{String} ] \Leftarrow m \text{ String} \]

\[ \text{words } = \text{union } (\text{nil } \circ \text{ ignore } "\" ) \leftarrow (\text{unfoldl } \circ \text{sepPut } "\") \]

\[ \text{unfoldl1 } \cdot\cdot\cdot \text{MonadExcept } m \Rightarrow ((a, a) \leftarrow m a) \rightarrow ([a] \Leftarrow m a) \]

\[ \text{unfoldl1 } f = \text{union } (\text{cons } \circ \text{inju (id } \otimes \text{unfoldl1 } f ) \circ \text{f } ) \rightarrow \text{wrap} \]

Our encoding mimics the standard Haskell definition of \textit{unwords}: it breaks the view string into two words separated by a single space, employing a custom \textit{sepPut}:(\text{String} , \Leftarrow m \text{ String}) putlens that fails if the string has no spaces; the empty view string generates an empty source list; and any other string (without spaces) is put to the source as a singleton word. The resulting putlens can be proved total for any source list and view string. For example:

\[ \text{get words } ["a", "b", "c"] = \text{Just } [a” \ b” \ c”] \]

\[ \text{put words } \text{Nothing } "he" "110" = \text{Just } ["he", "", "110"] \]

5. Implementation

This section unveils the implementation of our put-based lens language as a Haskell library. Although our presentation of putlenses this far focuses on writing put transformations, which is sufficient to describe the interface to users, our implementation internally manipulates explicit definitions of put. We also unveil a standard technique that improves the efficiency of our prototype implementation.

5.1 Explicit get functions with observable domains

First, we redefine a putlens as a record of a put and a get function:

\[ \text{data } s \Leftarrow m v = \text{Putlens } \{ \text{put } :: \text{Maybe } s \rightarrow v \rightarrow m s \}

\[ \text{get } :: s \rightarrow \text{Maybe } v \}

In this definition, we extend the view type of \textit{get} to \textit{Maybe \ v} in order to make its domain “observable”, i.e., for a particular source
Proposition 3. A put lens \( l :: s \leftarrow m, v \) is well-behaved iff it satisfies the following properties:

\[
\begin{align*}
\text{if } & s \in \text{put } s \quad \Rightarrow \quad \text{Just } v' = \text{get } s' & \text{PUTGet}_{\infty} \\
\text{Just } v \in \text{get } s & \Rightarrow \quad \text{put } (\text{Just } s) v = \text{return } s & \text{GETPut}_{\infty}
\end{align*}
\]

Figure 1 shows concrete get definitions for the combinators in our language. As an example, the get of addfst \( f \) always selects the second element of a pair, since we ensure that its put function always satisfies \( \text{PUTIdevice} \). A technical remark is that, for some combinators like remfst \( f \) or \( \Phi \), our implementation may overestimate the domain of get if their argument functions are not total. In practice, this does not disrupt well-behavedness and just renders the domain of get “non-observable” for some sources (get \( s = \text{Just } \bot \)); the domain of get is exact for total arguments.

5.2 Improve efficiency via tupling transformation

The traditional usage scenario of lenses is to apply get to an original source to compute a view, and invoke put with an updated view to produce a correspondingly updated source. Since put takes into account the original source information in order to propagate a view update, it often performs redundant computations of get — an elucidative example is putlenses composition \((\circ \cdot)\). For this reason, it is better to partially evaluate put for the original source at the same time we compute get; in algebraic programming, this optimization is known as tupling. In our implementation, we reshape the putlenses framework into an equivalent formulation but where get and put are tupled into a single function, as originally suggested in [27].

\[
\text{data } s \leftarrow m, v = \text{Putlens } \{ \text{create } v \rightarrow m s, \text{getput } s \rightarrow (\text{Maybe } v, v \rightarrow m s) \}
\]

A putlens in the tupled framework consists of two functions: a create function that corresponds to put for the case when the original source is not available, and a tupled getput function that takes an

\[\text{get id } s = \text{Just } s \quad \quad \quad \text{get } (f \circ g) s = \text{get } f s \triangleright\triangleright \text{ get } g\]

\[\text{get } (\Phi p) s = \text{if } p s \text{ then } \text{Just } s \text{ else Nothing}\]

\[\text{get } \text{bot } s = \text{Nothing} \quad \quad \quad \text{get } (\text{effect } f) s = \text{get } g s\]

\[\text{get } \text{addfst } f (s_1, s_2) = \text{Just } s_2 \quad \quad \quad \text{get } \text{addssnd } f (s_1, s_2) = \text{Just } s_1\]

\[\text{get } \text{remfst } f (s) = \text{Just } (f s, s) \quad \quad \quad \text{get } \text{inj } p (\text{Left } s) = \text{Just } s\]

\[\text{get } \text{remssnd } f (s) = \text{Just } (s, f s) \quad \quad \quad \text{get } \text{inj } p (\text{Right } s) = \text{Just } s\]

\[\text{get } (f \circ g) (s_1, s_2) = \text{do} \{ v_1 \leftarrow \text{get } s_1; v_2 \leftarrow \text{get } s_2; \text{return } (v_1, v_2) \}\]

\[\text{get } (f \circ g) s | \text{isNothing } (\text{get } f s) = \text{fmap Right } (\text{get } g s)\]

\[\text{get } (f \circ g) s | \text{isNothing } (\text{get } g s) = \text{fmap Left } (\text{get } f s)\]

\[\text{otherwise } = \text{Nothing}\]

\[\text{get } (f \circ g) (\text{Left } s_1) = \text{fmap Left } (\text{get } f s_1)\]

\[\text{get } (f \circ g) (\text{Right } s_2) = \text{fmap Right } (\text{get } g s_2)\]

\[\text{get in } s = \text{out } s \quad \quad \quad \quad \quad \text{get out } s = \text{inn } s\]
keepfst) and rather conservative—they live in the class of very well-behaved lenses [10]—having a much simpler bidirectional semantics. As evidence, our partial embedAt example can only be described as a traversal (which satisfies weaker laws unrelated to bidirectional behavior). The package also features type-changing view updates; this is fairly common for example when updating fields of polymorphic records, but becomes less tangible for more complex type-changing bidirectional transformations.

7. Conclusions

In this paper, we have proposed a novel put-based lens language that invites programmers to shift into a put programming style in which they write sort of injective put functions from view to source.

When writing lenses in traditional languages, users often have to think about the behavior of both the get and the put functions, but they may not be allowed to describe them in terms of the provided language constructs. Put-based programming is more powerful in that it allows them to specify a BX more precisely just in terms of put. The tradeoff is that some of the responsibility is passed from the bidirectional system to the lens programmer.

Notwithstanding, put-based lens programming is not necessarily more complex. Users can start with a (dual of a) get-oriented lens using default combinators. (It is worth noticing that most combinators in our language that have analogous in lens languages are derived ones, which supports the claim that our language is a proper superset of those languages.) Next, they can systematically adapt the behavior of unsatisfactory put lenses by combining monadic effects or replacing particular defaults with more general variants. Our examples suggest that this shift is manageable in practice, as the additional expressive power—although crucial to express intricate update strategies—is often needed only at precise parts of a BX. The additional complexity of writing a putlens can be roughly compared by ignoring monadic effects and user-defined parameters, to obtain a lens with essentially the same get but a different put.

To highlight the potential of put programming, we have developed a very flexible low-level put-based lens language, that we hope can serve as a general-purpose framework in which other BX approaches can be implemented. Our prototype library is available online at the Hackage package repository under the name putlenses, and bundles the examples presented in this paper and more. Nevertheless, it may be hard to write intricate put functions as put lenses, and the next step is to improve on static guarantees and programmability. On this account, we are currently developing a more maintainable (but less expressive) high-level put-based language for updating XML databases, with core semantics given as put lenses. In the future, a more serious user evaluation is necessary to assess the practical value of put programming techniques.

A further direction is to investigate the design of put programming languages for other data domains, e.g., relational databases.

Acknowledgments

We thank the NII’s Programming Research Laboratory for stimulating discussions on BXs and several anonymous reviewers for many insightful suggestions. This work is partly supported by Grant-in-Aid for Scientific Research (A) No. 25240009 in Japan.

References


