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-Albretchs 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 aid programmers in writing 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 get 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) [6] 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. [10] on a combinatorial language for bidirectional tree transformations named *lenses*. Lenses recast many of the ideas from *view updating* in the database community [1]. 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, denoted by $l :: s \Leftrightarrow v$, is a BX that comprises two (partial) functions $get :: s \rightarrow v$ and $put :: s \rightarrow v \rightarrow s$, satisfying the following properties:

$$v \in get\ s \;\;\Rightarrow\;\; s = put\ s\ v \qquad \textsc{GetPut}$$

$$s' \in put\ s\ v' \;\;\Rightarrow\;\; v' = get\ s' \qquad \textsc{PutGet}$$

A lens is said to be *total* if get and put are total functions. □

Here $y \in f\ x$ means that $f\ x$ is defined and that $y = f\ x$. Note that our properties are implications [18]: GETPUT ensures that a lens is *stable*, i.e., whenever a view produced by get is not modified, put must return the original source; PUTGET guarantees that a lens is *acceptable*, i.e., view updates must be translated exactly, so that (if put is defined) the updated view can be retrieved by applying 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 get function is in general not injective, so there may exist many possible put functions that combined with get form a well-behaved BX. Consider a get function that computes the height of a rectangle, written in Haskell [19] as $height\ (w, h) = h$. 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:

$$\begin{array}{l} put_{1\,height}\ (w, h)\ h' = (w, h') \\ put_{2\,height}\ (w, h)\ h' \mid w \equiv h = (h', h') \\ put_{3\,height}\ i\ (w, h)\ h' = (\mathbf{if}\ h \equiv h'\ \mathbf{then}\ w\ \mathbf{else}\ i, h') \end{array}$$

This unavoidable ambiguity of 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-

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2619-3/14/01…$15.00.
http://dx.doi.org/10.1145/2543728.2543737

ing approaches can be classified into two main methods. The first asks users to write get in a familiar (unidirectional) programming language, and derives one single suitable put through *bidirectionalization* techniques [13, 16, 17, 20, 28] (though some admit ad-hoc parameterization). This method has been mostly followed for view updating in the database community. The second method is to design a domain-specific *bidirectional programming language* whose programs can be interpreted both as a get function and a put function [4, 5, 10, 14, 23, 26]. This has been the favored method for functional bidirectional programming approaches, with various lens languages populating the spectrum of existing BX approaches. These languages tend to invite users to write lenses as they would write get functions, but providing eventually different put strategies via different combinators (that may accept additional parameters).

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 put functions using only a fixed set of strategies. For example, many existing bidirectional languages [10, 23] assume $put_{1_{height}}$ 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 [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 put. To change the programmer's mindset towards writing 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 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 put functions of well-behaved lenses and showing that these conditions are indeed sufficient to determine uniqueness of get. We explain our contributions in Sections 3, 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 [8, 9], 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 [22] or total bidirectional programming languages [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 $f :: a \rightarrow b$ can be defined by pattern matching over the input type a. For example, the $tail$ function from the standard Haskell prelude computes the tail of a list as follows.

$$tail\ (x : xs) = xs$$

Partiality can be implicitly modeled via non-exhaustive pattern matching, as for the empty list above. For non-defined patterns, we can say that $tail\ [\,]$ 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.:

$$heightSquare\ (w, h) \mid w \equiv h = h$$

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 $\mathsf{dom}(f)$. For two functions f and g, inclusion $f \sqsubseteq g$ is defined by $\forall a.\ b \in f\ a \Rightarrow b = g\ a$. 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 $get\ s$ is $\bot$ for a source s, then GETPUT does not restrict the result of $put\ s\ v'$ for arbitrary view values v'[1].

2.2 Put-based Laws

We now study certain implications of partial lens laws and use them to characterize partial lenses based on their put functions.

Proposition 1. *For a well-behaved lens, the function "$put\ s$" is* injective *for any source s, in the following sense:*

$$s' \in put\ s\ v \ \wedge\ s' \in put\ s\ v' \ \Rightarrow\ v = v' \qquad \text{PUTINJ}$$

Proposition 2. *For a well-behaved lens,* PUTTWICE *[10] holds:*

$$s' \in put\ s\ v \ \Rightarrow\ s' = put\ s'\ v \qquad \text{PUTTWICE}$$

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

[1] If $put\ s\ v'$ is defined and equal to s', then PUTGET requires $get\ s'$ to be defined and equal to v', 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* 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 $put\ s\ v = s$.
(b) A lens with a get function such that $get\ s = v \Leftrightarrow s = put\ s\ v$ 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 $put\ s\ v = s$. 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 $embedAt\ i$ function that embeds a view value at the i-th position of a source list, as illustrated in the call:

$$embedAt\ 2\ \texttt{"abcd"}\ \texttt{'x'} = \texttt{"abxd"}$$

The idea is to replace the element at position i in the source with the updated view value. In Haskell, we can write $embedAt$ as:

$$\begin{array}{l} embedAt :: Int \to [a] \to a \to [a] \\ embedAt\ 0\ (s : ss)\ v = v : ss \\ embedAt\ i\ (s : ss)\ v = s : embedAt\ (i-1)\ ss\ v \end{array}$$

The $embedAt$ 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 $embedAt\ i\ [\,]\ v$). It is not difficult to see that this partial put function is well-behaved: $embedAt\ i\ s$ is injective for all views whenever it is defined, and $embedAt\ i\ (embedAt\ i\ s\ v)\ v$ 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 $embedAt$ as the following $\mathsf{embedAt}$ partial put-based lens.

$$\begin{array}{l} \mathsf{embedAt} :: Int \to ([a] \Leftarrow a) \\ \mathsf{embedAt}\ 0 = \mathsf{unhead} \\ \mathsf{embedAt}\ i = \mathsf{untail} \circ\!< \mathsf{embedAt}\ (i-1) \end{array}$$

This higher-order function performs induction on the argument index to produce a lens that embeds a view value at a fixed source index[2]. The auxiliary combinators $\mathsf{unhead} = \mathsf{cons} \circ\!< \mathsf{keepsnd}$ and $\mathsf{untail} = \mathsf{cons} \circ\!< \mathsf{keepfst}$ update the head and tail of a source list, where $\circ\!<$ denotes point-free lens composition.

The unique get function derived by our language is Haskell's predefined (!!) operator[3], that selects a list element by its index.

Since $embedAt$ is partial, a call like $embedAt\ 2\ \texttt{"a"}\ \texttt{'x'}$ will fail; $\mathsf{embedAt}\ i$ is only total for source lists with length greater than i. Using our language, we can systematically adapt $\mathsf{embedAt}$ to support this extra case, while preserving the same get, by redefining:

$$\begin{array}{l} \mathsf{unhead}' = \mathsf{cons} \circ\!< \mathsf{keepsndOr}\ (\lambda v \to return\ [\,]) \\ \mathsf{untail}' \ \ = \mathsf{cons} \circ\!< \mathsf{keepfstOr}\ (\lambda vs \to headM\ vs) \end{array}$$

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

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, $put :: s \to v \to s$ 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 Theorem 2.1, there is an essential condition independent of source values —for any source s, $put\ s$ (of type $v \to s$) 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.

$$\begin{array}{rl} Put ::= & \mathsf{id} \mid Put \circ\!< Put \mid \Phi\ p \mid \mathsf{bot} \mid \mathsf{effect}\ f\ Put \mid \mathsf{in} \mid \mathsf{out} \\ \mid & \mathsf{addfst}\ f \mid \mathsf{addsnd}\ f \mid \mathsf{remfst}\ f \mid \mathsf{remsnd}\ f \mid Put \otimes Put \\ \mid & \mathsf{inj}\ p \mid Put\ \nabla\ Put \mid Put \oplus Put \end{array}$$

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 ∇); 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

[2] We are using a different Sans-Serif font to differentiate lenses from the unidirectional Haskell functions such as $embedAt$, as followed in Section 3.

[3] The type and description of (!!) and many other Haskell standard functions used in this paper can be found via `http://haskell.org/hoogle`.

notation and technical details are explained below. The bidirectional laws are those from Section 2.2, lifted to monads.

Definition 3.1 (Put-based lens)**.** A well-behaved put-based lens l (*putlens* for short) is a bidirectional transformation

$$\textbf{type}\ s \Leftarrow_m v = \mathit{Maybe}\ s \to v \to m\ s$$

satisfying the following properties:

$$s' \in l\ s\ v \ \wedge\ s' \in l\ s\ v' \ \Rightarrow\ v = v' \qquad \text{PUTINJ}_{\Leftarrow}$$

$$s' \in l\ s\ v \ \Rightarrow\ l\ s'\ v = \mathit{return}\ s' \qquad \text{PUTTWICE}_{\Leftarrow}$$

In terms of interface, a putlens is just a put function; we give the alias $\mathsf{put} :: \mathit{Monad}\ m \Rightarrow (s \Leftarrow_m v) \to (\mathit{Maybe}\ s \to v \to m\ s)$. It takes as input not an original source of type s but an optional source of type $\mathit{Maybe}\ s$, to account for the cases when an original source does not exist and a new source must be reconstructed directly from the updated view. It is a standard technique to extend lens languages with such a $\mathit{create} :: v \to s$ function [5, 23]. Each well-behaved putlens l has a unique get function given as:

$$\mathsf{get}\ s = v \ \Leftrightarrow\ s \in l\ (\mathit{Just}\ s)\ v \qquad \text{GET}_{\Leftarrow}$$

Concrete definitions for our language are given in Section 5. □

Monads Additionally, we enrich put with an arbitrary monad m, allowing the use of various sorts of computational effects. As demonstrated in Section 4, this will enable the encoding of more global update strategies that go beyond the typically local and unsatisfactory update strategies induced by combinatorial BX approaches, without having to design a specialized BX language. The notion of monad is captured by the following Haskell type class.

$$\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b \\
\quad \varnothing \qquad :: m\ a
\end{array}$$

The two essential 'return' and 'bind' operations denote identity and sequential composition of monadic actions, satisfying particular soundness laws. To account for partial put computations, we consider an additional 'fail' operation along the lines of [11]. We will often make use of three conventional monadic combinators:

$$\begin{array}{l}
(\gg) :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b \\
\mathit{mx} \gg \mathit{my} = \mathit{mx} \gg\!= (\lambda x \to \mathit{my}) \\
\mathit{guard} :: \mathit{Monad}\ m \Rightarrow \mathit{Bool} \to m\ () \\
\mathit{guard}\ b = \textbf{if}\ b\ \textbf{then}\ \mathit{return}\ ()\ \textbf{else}\ \varnothing \\
\mathit{mfilter} :: \mathit{Monad}\ m \Rightarrow (a \to \mathit{Bool}) \to m\ a \to m\ a \\
\mathit{mfilter}\ p\ \mathit{mx} = \textbf{do}\ \{x \leftarrow \mathit{mx}; \mathit{guard}\ (p\ x); \mathit{return}\ x\}
\end{array}$$

The $\gg$ combinator composes two monadic actions by ignoring the first result; *guard* succeeds if a boolean condition holds or fails otherwise; *mfilter* checks a predicate applied to the current value[4].

To inspect values inside a monad, we introduce a *monad membership relation* $x \in m$ denoting that "some execution of the computation of monad m produces a pure value x", satisfying two laws:

$$x \in \mathit{return}\ x \qquad \in\text{-ID}$$

$$y \in m \gg\!= f \ =\ (\exists x.\ x \in m\ \wedge\ y \in f\ x) \qquad \in\text{-COMP}$$

For particular monads, we can make this abstract notion more precise as $x \in m \equiv (\exists h.\ h\ m = x)$, 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 $\mathit{State}\ \mathit{st}$ monad (presented later in Section 4) we can define $x \in m \equiv (\exists \mathit{st}.\ \mathit{evalState}\ m\ \mathit{st} = x)$.

[4] The **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 $\forall a.\ f\ a \neq \varnothing$.

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.

semantic constraints
put :: Haskell type put = implementation

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 $2 = \{\mathit{True}, \mathit{False}\}$ corresponds to the Haskell primitive Bool type), we write A_p for $\{a \mid a \in A \wedge p\ a\}$. We define set-theoretic unit ($1 = \{()\}$), product ($A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$) and disjoint sum ($A + B = \{\mathit{Left}\ x \mid x \in A\} \cup \{\mathit{Right}\ y \mid y \in B\}$) types.

3.1 Basic combinators

The identity combinator simply replaces the view for the source. View-based filtering Φ defines subsets of the identity putlens[5]:

$\mathsf{id} \in V \Leftarrow_m V$
$\mathsf{id} :: v \Leftarrow_m v$ $\mathsf{id}\ s\ v' = \mathit{return}\ v'$

$\Phi\ V \in V \Leftarrow_m V$
$\Phi :: (v \to \mathit{Bool}) \to (v \Leftarrow_m v)$ $\Phi\ p\ s\ v' = \mathit{guard}\ (p\ v') \gg \mathit{return}\ v'$

Although Φ may seem semantically similar to 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 $\mathsf{bot} :: s \Leftarrow_m v$ 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:

$\dfrac{f \in S \Leftarrow_m U \quad g \in U \Leftarrow_m V}{f \circ\!< g \in S \Leftarrow_m V}$
$(\circ\!<) :: (s \Leftarrow_m u) \to (u \Leftarrow_m v) \to (s \Leftarrow_m v)$ $(f \circ\!< g)\ s\ v' = g\ (\mathit{fmap}\ (\mathsf{get}\ f)\ s)\ v' \gg\!= f\ s$

Composition first applies g (same as $\mathsf{put}\ g$) 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 $\mathsf{get}\ f$ function computes the original intermediate view of g, and is applied to the "maybe" original source using $\mathit{fmap} :: (a \to b) \to \mathit{Maybe}\ a \to \mathit{Maybe}\ b$ (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.

[5] For readability, we will often omit necessary Haskell type contexts like $\mathit{Monad}\ m$ or $\mathit{Eq}\ a$ 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:

$$\frac{f \in Maybe\ S \to V \to n \dot{\to} m \quad g \in S \Leftarrow_n V}{\mathsf{effect}\ f\ g \in S \Leftarrow_m V}$$

$\mathsf{effect} :: (Maybe\ s \to v \to n \dot{\to} m) \to (s \Leftarrow_n v) \to (s \Leftarrow_m v)$
$\mathsf{effect}\ f\ g\ s\ v' = f\ s\ v'\ (g\ s\ v')$

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

$$f\ s\ v'\ (return\ x) = return\ x$$

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 combinators —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 addfst combinator (and dual addsnd) creates a pair in the source by adding a new element to the left (or right) of the view:

$$\frac{P \subseteq S_1 \times V \qquad f \in Maybe\ P \to V \to m\ S_1 \qquad f\ (Just\ (s_1, v))\ v = return\ s_1}{\mathsf{addfst}\ f \in P \Leftarrow_m V}$$

$\mathsf{addfst} :: (Maybe\ (s_1, v) \to v \to m\ s_1) \to ((s_1, v) \Leftarrow_m v)$
$\mathsf{addfst}\ f = enforceGetPut\ put'$ **where**
$\quad put'\ s\ v' = f\ s\ v' \ggg= \lambda s_1' \to return\ (s_1', v')$

Its put 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 PUTTWICE$_\Leftarrow$ 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 $enforceGetPut$, defined later in Section 5. At the semantic level, we state a condition that user-defined functions shall statically satisfy to be well-behaved. The source domain of addfst f 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, 23] is the conservative variant of addfst that restores the original source if available:

$$\forall f \in V \to m\ S_1.\ \mathsf{keepfstOr}\ f \in S_1 \times V \Leftarrow_m V$$

$\mathsf{keepfstOr} :: (v \to m\ s_1) \to ((s_1, v) \Leftarrow_m v)$
$\mathsf{keepfstOr}\ f = \mathsf{addfst}\ (\lambda s\ v' \to maybe\ (f\ v')\ (return \circ fst)\ s)$

Like many other derived combinators that we will define, 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 keepfstOr does not introduce any dependency. A partial projection that only knows to recover the original source is given by $\mathsf{keepfst} = \mathsf{keepfstOr}\ (\lambda v \to \varnothing)$.

Another variant of addfst is to always 'copy' the updated view value in duplicate to create a new source pair:

$$\mathsf{copy} \in (V \times V)_{\overleftarrow{id}} \Leftarrow_m V$$

$\mathsf{copy} :: (v, v) \Leftarrow_m v$
$\mathsf{copy} = \Phi\ \overleftarrow{id} \circ\!\!< \mathsf{addfst}\ (\lambda s \to return)$

The copy putlens is only total for source pairs with equal components, as modeled by the predicate $\overleftarrow{id}$ (with $\overleftarrow{f}\ (x, y) = x \equiv f\ y$). The definition explicitly restricts its source domain[7]. The more relaxed $\mathsf{addfst}\ (\lambda s \to return)$ resembles the 'merge' lens from [10].

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

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

$$\forall f \in V \to V_1.\ \mathsf{remfst}\ f \in V \Leftarrow_m (V_1 \times V)_{\overleftarrow{f}}$$

$\mathsf{remfst} :: (v \to v_1) \to (v \Leftarrow_m (v_1, v))$
$\mathsf{remfst}\ f\ s\ (v_1', v') = guard\ (f\ v' \equiv v_1') \gg return\ v'$

To ensure PUTINJ$_\Leftarrow$, their put functions are only defined if the discarded view value can be reconstructed by applying a user-provided function to the kept value; the typing rules capture this dependency. Note that such argument function is not monadic, as it will be used in the get direction to reconstruct a new source.

The following laws emphasize the duality between our addition and removal combinators on products.

$$\Phi\ \overleftarrow{f} \circ\!\!< \mathsf{addfst}\ (\lambda s \to return \circ f) \circ\!\!< \mathsf{remfst}\ f \sqsubseteq \Phi\ \overleftarrow{f}$$
$$\mathsf{remfst}\ f \circ\!\!< \mathsf{addfst}\ (\lambda s \to return \circ f) \sqsubseteq \mathsf{id}$$

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:

$$\forall x \in V.\ \mathsf{ignore}\ x \in 1 \Leftarrow_m \{x\}$$

$\mathsf{ignore} :: v \to (() \Leftarrow_m v)$
$\mathsf{ignore}\ v = \mathsf{remfst}\ (\lambda() \to v)$
$\quad \circ\!\!< \mathsf{addsnd}\ (\lambda s\ v \to return\ ())$

$$\forall x \in S.\ \mathsf{new}\ x \in S \Leftarrow_m 1$$

$\mathsf{new} :: s \to (s \Leftarrow_m ())$
$\mathsf{new}\ x = \mathsf{remfst}\ (\lambda s \to ())$
$\circ\!\!< \mathsf{addsnd}\ (\lambda s\ v \to return\ x)$

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].

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

$$\frac{f \in S_1 \Leftarrow_m V_1 \qquad g \in S_2 \Leftarrow_m V_2}{f \otimes g \in S_1 \times S_2 \Leftarrow_m V_1 \times V_2}$$

$(\otimes) :: (s_1 \Leftarrow_m v_1) \to (s_2 \Leftarrow_m v_2) \to ((s_1, s_2) \Leftarrow_m (v_1, v_2))$
$(f \otimes g)\ s\ (v_1', v_2') = \mathbf{do}\ \{ s_1' \leftarrow f\ (fmap\ fst\ s)\ v_1';$
$\quad s_2' \leftarrow g\ (fmap\ snd\ s)\ v_2'; return\ (s_1', s_2') \}$

[6] We could instead define a weaker version of addfst f receiving a non-monadic f function and enforcing PUTTWICE$_\Leftarrow$ by making get partial.

[7] The backward behavior of addfst would otherwise enforce PUTTWICE$_\Leftarrow$ for any source pair, as it is not able to infer the more precise source type.

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 *Identity* or *Reader* defined later) f and g can be computed in parallel. Its non-monadic variant is known as the 'product' [14, 23] or 'concatenation' [5] lens.

3.4 Sums

Moving to sums, the 'injection' combinator uses a predicate to decides whether to "tag" views as left or right values in the source:

$$\frac{\begin{array}{c} p \in \mathit{Maybe}\ (V_1 + V_2) \to V_1 \cup V_2 \to m\ 2 \\ p\ (\mathit{Just}\ (\mathit{Left}\ v))\ v = \mathit{return}\ \mathit{True} \\ p\ (\mathit{Just}\ (\mathit{Right}\ v))\ v = \mathit{return}\ \mathit{False} \end{array}}{\mathsf{inj}\ p \in V_1 + V_2 \Leftarrow_m V_1 \cup V_2}$$

$$\begin{array}{l} \mathsf{inj} :: \ (\mathit{Maybe}\ (\mathit{Either}\ v\ v) \to v \to m\ \mathit{Bool}) \\ \quad \to (\mathit{Either}\ v\ v \Leftarrow_m v) \\ \mathsf{inj}\ p = \mathit{enforceGetPut}\ \mathit{put}'\ \mathbf{where}\ \mathit{put}'\ s\ v' = p\ s\ v' \gg\!= \\ \quad \lambda b \to \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ (\mathit{Left}\ v')\ \mathbf{else}\ \mathit{return}\ (\mathit{Right}\ v') \end{array}$$

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

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

$$\forall p \in V \to m\ 2.\ \mathsf{injsOr}\ p \in V + V \Leftarrow_m V$$

$$\begin{array}{l} \mathsf{injsOr} :: (v \to m\ \mathit{Bool}) \to (\mathit{Either}\ v\ v \Leftarrow_m v) \\ \mathsf{injsOr}\ p = \mathsf{inj}\ (\lambda s\ v' \to \mathit{maybe}\ (p\ v')\ (\mathit{return} \circ \mathit{isLeft})\ s) \\ \quad \mathbf{where}\ \mathit{isLeft} = \mathit{either}\ (\mathit{const}\ \mathit{True})\ (\mathit{const}\ \mathit{False}) \end{array}$$

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 [24].

The 'either' combinator (∇) enables the specification of putlenses by case analysis and applies two different putlenses depending on the view branching, producing a source of the same type:

$$\frac{f \in S_1 \Leftarrow_m V_1 \quad g \in S_2 \Leftarrow_m V_2 \quad S_1 \cap S_2 = \emptyset}{f \nabla g \in S_1 \cup S_2 \Leftarrow_m V_1 + V_2}$$

$$\begin{array}{l} (\nabla) :: (s \Leftarrow_m v_1) \to (s \Leftarrow_m v_2) \to (s \Leftarrow_m \mathit{Either}\ v_1\ v_2) \\ (f \nabla g)\ s\ (\mathit{Left}\ v_1') = \mathit{mfilter}\ (\mathit{disjoint}\ f\ g)\ (f\ v_1') \\ (f \nabla g)\ s\ (\mathit{Right}\ v_2') = \mathit{mfilter}\ (\mathit{disjoint}\ g\ f)\ (g\ v_2') \\ \mathit{disjoint}\ x\ y\ s = s \in \mathsf{dom}(\mathsf{get}\ x) \wedge s \notin \mathsf{dom}(\mathsf{get}\ y) \end{array}$$

It applies put f or put g to left or right view values, respectively. To guarantee $\text{PUTINJ}_{\Leftarrow}$, 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 *disjoint*, whose particular implementation is discussed later in Section 5. Accordingly, the typing rule requires the source domains of f and g to be disjoint.

One way to guarantee disjointness for ∇ is to ask programmers to provide a predicate that declares how to branch source values:

$$f\ \nabla_p\ g = (\Phi\ p \circ\!< f)\ \nabla\ (\Phi\ (\mathit{not} \circ p) \circ\!< g)$$

Here, the Φ combinator restricts the left and right source domains according to the predicate. We can also define left- ($f \mathbin{\dot{\nabla}} g$) and right-biased ($f \mathbin{\nabla\!\!\!\cdot}\ g$) variants of ∇_{S_1} that favor the domain of the left or right putlens, by instantiating $S_1 = \mathsf{dom}(f)$ or $S_1 = \neg(\mathsf{dom}(g))$.

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

$$(\mathsf{id}\ \nabla_p\ \mathsf{id}) \circ\!< \mathsf{inj}\ (\lambda s \to \mathit{return} \circ p) \sqsubseteq \mathsf{id}$$
$$\Phi\ (p?) \circ\!< \mathsf{inj}\ (\lambda s \to \mathit{return} \circ p) \circ\!< (\mathsf{id}\ \nabla_p\ \mathsf{id}) \sqsubseteq \Phi\ (p?)$$

Specifically, injecting view tags with a predicate p and ignoring them with ∇_p or ignoring view tags with ∇_p and re-injecting them according to p are both subsets of the identity putlens. 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 $p? = \mathit{either}\ p\ (\mathit{not} \circ p)$).

The 'sum' combinator (as found in [14, 23]) applies two putlenses to distinct sides of the view, keeping the view branching:

$$\frac{f \in S_1 \Leftarrow_m V_1 \quad g \in S_2 \Leftarrow_m V_2}{f \oplus g \in S_1 + S_2 \Leftarrow_m V_1 + V_2}$$

$$\begin{array}{l} (\oplus) :: (s_1 \Leftarrow_m v_1) \to (s_2 \Leftarrow_m v_2) \\ \quad \to (\mathit{Either}\ s_1\ s_2 \Leftarrow_m \mathit{Either}\ v_1\ v_2) \\ (f \oplus g)\ s\ (\mathit{Left}\ v_1') = f\ (l\ s)\ v_1' \gg\!= \mathit{return} \circ \mathit{Left}\ \mathbf{where} \\ \quad l = \mathit{maybe}\ \mathit{Nothing}\ (\mathit{either}\ \mathit{Just}\ (\mathit{const}\ \mathit{Nothing})) \\ (f \oplus g)\ s\ (\mathit{Right}\ v_2') = f\ (r\ s)\ v_2' \gg\!= \mathit{return} \circ \mathit{Right}\ \mathbf{where} \\ \quad r = \mathit{maybe}\ \mathit{Nothing}\ (\mathit{either}\ (\mathit{const}\ \mathit{Nothing})\ \mathit{Just}) \end{array}$$

The putlenses 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 putlenses:

$$\mathsf{injl} \in V + \emptyset \Leftarrow_m V$$

$$\begin{array}{l} \mathsf{injl} :: \mathit{Either}\ v\ v_2 \Leftarrow_m v \\ \mathsf{injl} = (\mathsf{id} \oplus \mathsf{bot}) \circ\!< \mathsf{inj} \\ \quad (\lambda s\ v' \to \mathit{return}\ \mathit{True}) \end{array}$$

$$\mathsf{injr} \in \emptyset + V \Leftarrow_m V$$

$$\begin{array}{l} \mathsf{injr} :: \mathit{Either}\ v_1\ v \Leftarrow_m v \\ \mathsf{injr} = (\mathsf{bot} \oplus \mathsf{id}) \circ\!< \mathsf{inj} \\ \quad (\lambda s\ v' \to \mathit{return}\ \mathit{False}) \end{array}$$

When applied to the view, injl (and injr) injects the view with a left (or right) tag; bot generalizes the opposite choice to any type.

3.5 Recursion

In the point-free style of programming [3], algebraic data types are usually seen as sums of products. Each data type A comes equipped with an isomorphism $\mathit{out} \in A \to F\ A$ that exposes its top-level structure (in a sense, encoding pattern matching over that type), and its converse $\mathit{inn} \in F\ A \to A$ 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 $F\ A = 1 + A \times [A]$ and the following instances:

$$\begin{array}{l} \mathit{inn}\ s = \mathit{either}\ (\lambda() \to [\,])\ (\lambda(x, xs) \to x : xs)\ s \\ \mathit{out}\ l = \mathbf{case}\ l\ \mathbf{of}\ \{[\,] \to \mathit{Left}\ (); (x : xs) \to \mathit{Right}\ (x, xs)\} \end{array}$$

The *inn* and *out* isomorphisms can be lifted to the putlenses $\mathsf{in} \in A \Leftarrow_m F\ A$ and $\mathsf{out} \in F\ A \Leftarrow_m A$ in a trivial way. The usual constructors and deconstructors for lists are defined as follows.

$$\begin{array}{ll} \mathsf{nil} \ \ = \mathsf{in} \circ\!< \mathsf{injl} & \mathsf{unnil} \ \ = (\mathsf{id} \nabla \mathsf{bot}) \circ\!< \mathsf{out} \\ \mathsf{cons} = \mathsf{in} \circ\!< \mathsf{injr} & \mathsf{uncons} = (\mathsf{bot} \nabla \mathsf{id}) \circ\!< \mathsf{out} \end{array}$$

The putlenses nil and unnil construct/destruct an empty list, and cons and uncons construct/destruct a non-empty list.

Instead of introducing an explicit fixed-point combinator [3], we define recursive putlenses implicitly, relying on Haskell's lazy recursion mechanism. Consequently, it is not guaranteed that (the recursive functions of) recursive putlenses terminate, and ensuring totality (and well-behavedness[8]) requires tools beyond composi-

[8] Precisely speaking, our well-behavedness laws also demand put and get to be defined (i.e., terminating) for particular domains.

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 [7]. 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 map function can be lifted to a (total) recursive combinator that maps an argument (total) putlens over a view list:

$$\mathsf{map} :: (b \Leftarrow_m a) \to ([b] \Leftarrow_m [a])$$
$$\mathsf{map}\ f = \mathsf{in} \circ\!< (\mathsf{id} \oplus (f \otimes \mathsf{map}\ f)) \circ\!< \mathsf{out}$$

We can also define a putlens for the ubiquitous $foldr$ function:

$$\mathsf{unfoldr} :: ((b, a) \Leftarrow_m a) \to a \to ([b] \Leftarrow_m a)$$
$$\mathsf{unfoldr}\ f\ x = \mathsf{in} \circ\!< iterate \circ\!< (\mathsf{inj}\ (\lambda s \to return \circ (\equiv x)))$$
$$\quad \mathbf{where}\ iterate = \mathsf{ignore}\ x \oplus ((\mathsf{id} \otimes \mathsf{unfoldr}\ f\ x) \circ\!< f)$$

The unfoldr putlens[9] takes a putlens f and a default view value x. Its get function will perform just as the (uncurried) $foldr$ function, and fold a list of type $[b]$ in a bottom-up approach to produce a value of type a, by applying get 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 inj 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, unfoldr will only build a total well-behaved putlens if the given put 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 $put\ \ s = f$ and $get = f^{-1}$. In existing combinatorial BX languages [21, 23], the variable-free style used to specify BXs requires making the control flow explicit through the use of some "piping" isomorphisms. These combinators play an important role in extending the expressiveness of the bidirectional language (as lens categories [14, 24] do not support categorical sums and products). For example, the following piping combinators reflect the commutativity, associativity and distributivity of sums and products.

$$\begin{array}{ll}
\mathsf{swap} & :: (b, a) \Leftarrow_m (a, b) \\
\mathsf{assocl} & :: ((a, b), c) \Leftarrow_m (a, (b, c)) \\
\mathsf{coswap} & :: Either\ b\ a \Leftarrow_m Either\ a\ b \\
\mathsf{coassocl} & :: Either\ (Either\ a\ b)\ c \Leftarrow_m Either\ a\ (Either\ b\ c) \\
\mathsf{distl} & :: Either\ (a, c)\ (b, c) \Leftarrow_m (Either\ a\ b, c) \\
\mathsf{undistl} & :: (Either\ a\ b, c) \Leftarrow_m Either\ (a, c)\ (b, c)
\end{array}$$

The putlens inverses for these combinators are correspondingly named assocr, coassocl, distr and 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:

$$\mathsf{ifthenelse}\ p\ f\ g = (f \mathbin{\dot\nabla} g) \circ\!< \mathsf{inj}\ p$$

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 inj and $\dot\nabla$: source domains must be disjoint and view domains can overlap.

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

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:

$$\frac{V_1 \subseteq V \quad f \in S_1 \Leftarrow_m V_1 \quad g \in S \setminus S_1 \Leftarrow_m V \setminus V_1}{\mathsf{ifVthenelse}\ V_1\ f\ g \in S \Leftarrow_m V}$$

$$\mathsf{ifVthenelse} :: (v \to Bool) \to (s \Leftarrow_m v) \to (s \Leftarrow_m v)$$
$$\quad \to (s \Leftarrow_m v)$$
$$\mathsf{ifVthenelse}\ p\ f\ g = ((f \circ\!< \Phi\ p) \mathbin{\dot\nabla} g) \circ\!< \mathsf{inj}\ (\lambda s \to return \circ p)$$

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 $acond$ combinator from [10].

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

$$\frac{S_1 \subseteq S \quad f \in S_1 \Leftarrow_m V \quad g \in S \setminus S_1 \Leftarrow_m V}{\mathsf{ifSthenelse}\ S_1\ f\ g \in S \Leftarrow_m V}$$

$$\mathsf{ifSthenelse} :: (s \to Bool) \to (s \Leftarrow_m v) \to (s \Leftarrow_m v)$$
$$\quad \to (s \Leftarrow_m v)$$
$$\mathsf{ifSthenelse}\ p\ f\ g = (f \nabla_p g) \circ\!< \mathsf{injsOr}\ (\lambda v \to return\ True)$$

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 injsOr, the view domains of f and g must be identical. This corresponds to the $ccond$ 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 map.

$$\mathsf{map}\ f = \mathsf{ifVthenelse}\ null\ (\mathsf{nil} \circ\!< \mathsf{unnil})\ iterate$$
$$\quad \mathbf{where}\ iterate = \mathsf{cons} \circ\!< (f \otimes \mathsf{map}\ f) \circ\!< \mathsf{uncons}$$

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 put and get functions) can be encoded using custom:

$$\mathsf{custom} :: (Maybe\ s \to v \to m\ s) \to (s \to v) \to (s \Leftarrow_m v)$$
$$\mathsf{custom}\ put\ get = \mathsf{remfst}\ get \circ\!< \mathsf{addsnd}\ (\lambda p \to put\ (fmap\ snd\ p))$$

Custom putlenses are always well-behaved[10], but should only be used for simple transformations since programmers must write two put and 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.

[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 $runIdentity :: Identity\ a \to a$ 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 $s \Leftarrow v$ as short for $s \Leftarrow_{Identity} v$.

The $Reader\ r$ monad models computations that access a shared environment r. We can run a reader-aware putlens $l :: s \Leftarrow_{Reader\ r} v$ 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, for example:

$$\begin{array}{l} \mathsf{runReader} :: (Maybe\ s \to v \to m\ r) \\ \qquad \to (s \Leftarrow_{ReaderT\ r\ m} v) \to (s \Leftarrow_m v) \\ \mathsf{runReader}\ f\ g\ s\ v' = \mathsf{effect}\ (\lambda s\ v' \to f\ s\ v' \ggg= runReaderT\ m) \end{array}$$

The runReader combinator runs a putlens computation in a newly initialized environment[11]. For the sake of generalization, the $ReaderT$ monad transformer ($Reader\ r = ReaderT\ r\ Identity$) adds a reader environment to any other monad.

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

$$\begin{array}{l} \mathsf{runState} :: (Maybe\ s \to v \to m\ st) \\ \qquad \to (s \Leftarrow_{StateT\ st\ m} v) \to (s \Leftarrow_m v) \end{array}$$

A particular subclass of monads that permit computations to recover from failure is captured by the *exception* monad [11]:

$$\begin{array}{l} \mathbf{class}\ Monad\ m \Rightarrow MonadExcept\ m\ \mathbf{where} \\ \quad catch :: m\ a \to m\ a \to m\ a \end{array}$$

The intention is that $\varnothing$ 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 $fromJust :: Maybe\ a \to a$.

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 [12] or for lenses that are non-deterministic in general [18].

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 unfoldr from Section 3. 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:

$$summands :: [Int] \to Int \to [Int]$$

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 $summands$ putlenses.

$$\begin{array}{ll} > get\ [1,2,3,4] & > put_1\ [1,2,3,4]\ 15 \\ 10 & [1,2,3,4,5] \\ > put_2\ [1,2,3,4]\ 15 & > put_3\ [1,2,3,4]\ 15 \\ [3,3,4,4,1] & [2,3,4,6] \end{array}$$

The get function is the same for the three putlenses: the Haskell sum function that sums a list of numbers. Our first strategy (put_1) 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 (put_2) 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 (put_3) 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 putlens that splits a view integer into two summands[12]:

$$\begin{array}{l} \mathsf{split} :: (Int \to Int \to m\ Int) \to ((Int, Int) \Leftarrow_m Int) \\ \mathsf{split}\ offset = \mathsf{custom}\ put\ (uncurry\ (+))\ \mathbf{where} \\ \quad put\ (Just\ (x,y))\ z = \mathbf{do}\ i \leftarrow offset\ (x+y)\ z \\ \qquad\qquad\qquad\qquad\qquad return\ (x+i, z-x-i) \\ \quad put\ Nothing\ z = return\ (z, 0) \end{array}$$

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 (put_1, put_2, put_3) for $summands$ using a recursive unfoldr and different offsets:

$$\begin{array}{l} \mathsf{summands}_1, \mathsf{summands}_2, \mathsf{summands}_3 :: [Int] \Leftarrow_m Int \\ \mathsf{summands}_1 = \mathsf{unfoldr}\ (\mathsf{split}\ (\lambda s\ v \to return\ 0))\ 0 \\ \mathsf{summands}_2 = \mathsf{unfoldr}\ (\mathsf{split}\ (\lambda v\ v' \to div\ (v'-v)\ 2))\ 0 \\ \mathsf{summands}_3 = \mathsf{runState}\ (\lambda s\ v \to len\ s)\ (\mathsf{unfoldr}\ split\ 0) \\ \quad \mathbf{where}\ len = return \circ maybe\ 0\ length \\ \qquad\qquad split = \mathsf{split}\ (\lambda v\ v' \to readState \ggg= \lambda i \to \\ \qquad\qquad\quad writeState\ (i-1) \gg return\ (div\ (v'-v)\ i)) \end{array}$$

The put of $\mathsf{summands}_1$ 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 $\mathsf{summands}_2$, split divides the view update in half. To evenly split the view update, $\mathsf{summands}_3$ 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.

$$\begin{array}{ll} isort :: Ord\ a \Rightarrow [a] \to a & ins :: Ord\ a \Rightarrow a \to [a] \to [a] \\ isort\ [] = [] & ins\ x\ [] = [x] \\ isort\ (x:xs) = & ins\ x\ (y:ys) = \mathbf{if}\ x \leqslant y \\ \quad ins\ x\ (isort\ xs) & \quad \mathbf{then}\ x:y:ys\ \mathbf{else}\ y : ins\ x\ ys \end{array}$$

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:

[11] Remember that for, runReader to be well-behaved, programmers must ensure that $f\ (Just\ s)\ (get\ s) = return\ r$, for some environment r.

[12] Note that custom will enforce PUTTWICE for any $offset$ function.

put_1 $[4, 1, 3, 2]$ $[5, 6, 7, 8, 9, 10] = [8, 5, 7, 6, 9, 10]$
put_2 $[4, 1, 3, 2]$ $[5, 6, 7, 8, 9, 10] = [5, 6, 7, 8, 9, 10]$
put_2 $[4, 1, 3, 2]$ $[1, 2, 3, 4] = [4, 1, 3, 2]$

Our first strategy ($\mathsf{iunsort_1}$) 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 $\mathsf{iunsort_2}$ that simply copies the (sorted) view to the source whenever the view is modified.

We can define the first putlens by dualizing $isort$ to $\mathsf{iunsort_1}$ and ins to del using default conditional combinators as follows.

$\mathsf{iunsort_1} :: Ord\ a \Rightarrow ([a] \Leftarrow_m [a])$
$\mathsf{iunsort_1} = \mathsf{ifVthenelse}\ null\ (\mathsf{nil} \circ\!\!< \mathsf{unnil})\ iterate$
  $\textbf{where}\ iterate = \mathsf{cons} \circ\!\!< (\mathsf{id} \otimes \mathsf{iunsort_1}) \circ\!\!< \mathsf{del}$

$\mathsf{del} = \mathsf{ifVthenelse}\ (null \circ snd)\ \mathsf{id}\ disarrange \circ\!\!< \mathsf{uncons}$
  $\textbf{where}\ disarrange = \mathsf{ifSthenelse}\ p\ \mathsf{id}\ reorder$
    $p\ (x, y : ys) = x \leqslant y$
    $reorder = (\mathsf{id} \otimes \mathsf{cons}) \circ\!\!< \mathsf{subr} \circ\!\!< (\mathsf{id} \otimes \mathsf{del})$

The piping combinator subr (that swaps the order of x and y in the code of ins) is defined as $\mathsf{assocr} \circ\!\!< (\mathsf{swap} \otimes \mathsf{id}) \circ\!\!< \mathsf{assocl}$. The definition of $\mathsf{iunsort_1}$ 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 $\mathsf{ifSthenelse}$ 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., $\mathsf{iunsort_2}$ is defined using $disarrange = \mathsf{ifthenelse}\ mod\ (\Phi\ p)\ reorder$, with $mod\ (Just\ (Right\ v))\ v = return\ False$ and $mod\ s\ v = return\ True$ otherwise. As expected, both $\mathsf{iunsort_1}$ and $\mathsf{iunsort_2}$ 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 [6]), the put function of a lens must align the original source with the modified view to identity view modifications and translate them to the source. This *alignment* problem is well-known in the literature, and some languages [2, 5, 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:

$\textbf{data}\ Person = Person\ \{name :: Name, city :: City\}$
$\textbf{type}\ Name = String \qquad \textbf{type}\ City = String$

$people = [peter, roberto, david]$
$peter = Person$ `"Peter" "San Diego"`
$roberto = Person$ `"Roberto" "Rome"`
$david = Person$ `"David" "San Diego"`

The following putlenses allow modifying the fields of people.

$\mathsf{name} :: Person \Leftarrow_m Name \qquad \mathsf{city} :: Person \Leftarrow_m City$
$\mathsf{name} = \mathsf{in} \circ\!\!< \mathsf{keepsnd} \qquad \mathsf{city} = \mathsf{in} \circ\!\!< \mathsf{keepfst}$

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.

$\mathsf{peopleNames_0} : City \to ([Person] \Leftarrow_m [Name])$
$\mathsf{peopleNames_0}\ newc = \mathsf{map}\ (\mathsf{in} \circ\!\!< \mathsf{addsnd}\ cityOf)$
  $\textbf{where}\ cityOf\ s\ v = return\ (maybe\ newc\ snd\ s)$

This projection putlens maps $\mathsf{addsnd}\ cityOf$ 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:

$> \textbf{let}\ s\ = [roberto, david]$
$> \textbf{let}\ v' = [$`"Peter"`, `"Roberto"`, `"David"`$]$
$> runIdentity\ (\mathsf{put}\ (\mathsf{peopleNames_0}$ `"Lyon"`$)\ s\ v')$
$[Person$ `"Peter" "Rome"`, $Person$ `"Roberto" "San Diego"`
, $Person$ `"David" "Lyon"`$]$

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):

$\mathsf{peopleNames} :: City \to [Person] \Leftarrow_{Reader\ [Person]} [Name]$
$\mathsf{peopleNames}\ newc = \mathsf{map}\ (\mathsf{in} \circ\!\!< \mathsf{addsnd}\ cityOf)\ \textbf{where}$
  $cityOf\ s\ n = ask \ggg\!\!= \lambda people \to return$
    $(\textbf{case}\ find\ (\lambda p \to get\ \mathsf{name}\ p \equiv n)\ people\ \textbf{of}$
      $\{Just\ p \to get\ \mathsf{city}\ p; Nothing \to newc\})$

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

$> runReader\ (\mathsf{put}\ (\mathsf{peopleNames}$ `"Lyon"`$)\ s\ v')\ s$
$[Person$ `"Peter" "Lyon"`, $roberto, david]$ □

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 [2, 5, 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 $peopleFrom\ from$ that selects people from a city $from$, and two update strategies: $\mathsf{peopleFrom}\ from$ that removes deleted people in the view, and $\mathsf{peopleFromTo}\ from\ to$ that moves deleted people to a new city to different than $from$:

$> peopleFrom$ `"San Diego"` $people = [peter, david]$
$>$ $\mathsf{put}\ (\mathsf{peopleFrom}$ `"San Diego"`$)\ people\ [david]$
$Identity\ [roberto, david]$
$>$ $\mathsf{put}\ (\mathsf{peopleFromTo}$ `"San Diego"` `"Lyon"`$)\ people\ [david]$
$Identity\ [Person$ `"Peter"` `"Lyon"`$, roberto, david]$

We start by defining a general selection putlens that manipulates an internal state of filtered out rows to be recovered:

$$
\begin{array}{l}
\mathsf{select} :: Ord\ k \Rightarrow (a \to k) \to (Maybe\ [a] \to m\ [a]) \\
\quad\quad \to (a \to Bool) \to ([a] \Leftarrow_m [a]) \\
\mathsf{select}\ key\ entries\ p = \mathsf{runState}\ rs\ (\mathsf{select}'\ key\ p)\ \mathbf{where} \\
\quad rs\ s\ v = entries\ s \ggg \lambda rs \to return\ (Nothing, rs) \\
\mathsf{select}'\ key\ p = \mathsf{ifthenelse}\ cond\ recover\ iterate\ \mathbf{where} \\
\quad cond\ s\ v' = \mathbf{do}\ (\_, rs) \leftarrow readState \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ (h, t) = recoverRow\ key\ rs\ v' \\
\quad\quad\quad\quad\quad\quad\quad writeState\ (h, t) \gg return\ (isJust\ h) \\
\quad recover = \mathsf{cons} \propto (\Phi\ (not \circ p) \otimes \mathsf{select}'\ key\ p) \propto \mathsf{addfst} \\
\quad\quad (\lambda s\ v \to \mathbf{do}\ \{(Just\ x, \_) \leftarrow readState; return\ x\}) \\
\quad iterate = \mathsf{in} \propto (\mathsf{id} \oplus \mathsf{id} \otimes \mathsf{select}'\ key\ p) \propto \mathsf{out} \\
recoverRow\ key\ [\,]\ \_ = (Nothing, [\,]) \\
recoverRow\ key\ (x : xs)\ [\,] = (Just\ x, xs) \\
recoverRow\ key\ (x : xs)\ (v : vs) \\
\quad \mid key\ v < key\ x = (Nothing, x : xs) \\
\quad \mid key\ v \equiv key\ x = (Nothing, xs) \\
\quad \mid key\ v > key\ x = (Just\ x, xs)
\end{array}
$$

Our select combinator takes as arguments a predicate p (the filtering criterion), a key function that uniquely determines rows, and an $entries$ function that computes a subtable of rows (not satisfying p) 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 $\mathsf{peopleFrom}$ by initializing the state of select with only the people not in $from$, and $\mathsf{peopleFromTo}$ by considering also the people originally in city $from$ but moved to city to:

$$
\begin{array}{l}
\mathsf{peopleFrom} :: City \to ([Person] \Leftarrow_m [Person]) \\
\mathsf{peopleFrom}\ from = \mathsf{select}\ (get\ \mathsf{name})\ rs\ (isFrom\ from) \\
\quad\quad \mathbf{where}\ rs = return \circ maybe\ [\,]\ (filter\ (not \circ isFrom\ from)) \\
\mathsf{peopleFromTo} :: City \to City \to ([Person] \Leftarrow_m [Person]) \\
\mathsf{peopleFromTo}\ from\ to = \mathsf{select}\ (get\ \mathsf{name})\ rs\ (isFrom\ from) \\
\quad \mathbf{where}\ rs = return \circ maybe\ [\,]\ (map\ move) \\
\quad\quad move\ p = \mathbf{if}\ isFrom\ from\ p\ \mathbf{then}\ put\ \mathsf{city}\ p\ to\ \mathbf{else}\ p
\end{array}
$$

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

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 [4, 8]. 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 $catch$, we can express this behavior as a 'union' putlens:

$$
\frac{f \in S_1 \Leftarrow_m V_1 \qquad g \in S \setminus S_1 \Leftarrow_m V_2}{\mathsf{union}\ f\ g \in S \Leftarrow_m V_1 \cup V_2}
$$

$$
\begin{array}{l}
\mathsf{union} :: (s \Leftarrow_m v) \to (s \Leftarrow_m v) \to (s \Leftarrow_m v) \\
\mathsf{union}\ f\ g = (\Phi\ (\mathsf{dom}(\mathsf{get}\ f))\ \nabla\ \mathsf{id}) \propto \mathsf{inju}\ \mathbf{where} \\
\quad \mathsf{inju}\ Nothing\ v = (f\ Nothing\ v \ggg return \circ Left) \\
\quad\quad `catch`\ (g\ Nothing\ v \ggg return \circ Right) \\
\quad \mathsf{inju}\ (Just\ (Left\ s))\ v = (f\ (Just\ s)\ v \ggg return \circ Left) \\
\quad\quad `catch`\ (g\ Nothing\ v' \ggg return \circ Right) \\
\quad \mathsf{inju}\ (Just\ (Right\ s))\ v = (g\ (Just\ s)\ v \ggg return \circ Right) \\
\quad\quad `catch`\ (f\ Nothing\ v \ggg return \circ Left)
\end{array}
$$

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 $\mathsf{ifthenelse}$, but in a less effective way.) The putback direction applies f or g, preferring f when the view domains overlap except if the original source belongs to the source domain of g. It can be found in the literature as the $cond$ [10] or 'union' [5] lens. The 'union' lens of [14] 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 $unwords$ 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:

$$
\begin{array}{l}
\mathsf{words} :: MonadExcept\ m \Rightarrow [String] \Leftarrow_m String \\
\mathsf{words} = \mathsf{union}\ (\mathsf{nil} \propto \mathsf{ignore}\ \texttt{""})\ (\mathsf{unfoldr1}\ (\mathsf{sepPut}\ \texttt{" "})) \\
\mathsf{unfoldr1} :: MonadExcept\ m \Rightarrow ((a, a) \Leftarrow_m a) \to ([a] \Leftarrow_m a) \\
\mathsf{unfoldr1}\ f = \mathsf{union}\ (\mathsf{cons} \propto (\mathsf{id} \otimes \mathsf{unfoldr1}\ f) \propto f)\ \mathsf{wrap}
\end{array}
$$

Our encoding mimics the standard Haskell definition of $unwords$: it breaks the view string into two words separated by a single space, employing a custom $\mathsf{sepPut} :: ((String, String) \Leftarrow_m String)$ 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:

$\mathsf{get\ words}$ `["a","b","c"]` $= Just$ `"a b c"`
$\mathsf{put\ words}\ Nothing$ `"he  llo"` $= Just$ `["he","","llo"]` □

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 get. 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:

$$
\begin{array}{l}
\mathbf{data}\ s \Leftarrow_m v = Putlens\ \{\mathsf{put} :: Maybe\ s \to v \to m\ s \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad , \mathsf{get} :: s \to Maybe\ v\}
\end{array}
$$

In this definition, we extend the view type of get to $Maybe\ v$ in order to make its domain "observable", i.e., for a particular source

$$
\begin{array}{ll}
\mathsf{get}\ \mathsf{id}\ s = \mathit{Just}\ s & \mathsf{get}\ (f \circ\!\!< g)\ s = \mathsf{get}\ f\ s \gg\!\!= \mathsf{get}\ g \\
\multicolumn{2}{l}{\mathsf{get}\ (\Phi\ p)\ s = \mathbf{if}\ p\ s\ \mathbf{then}\ \mathit{Just}\ s\ \mathbf{else}\ \mathit{Nothing}} \\
\mathsf{get}\ \mathsf{bot}\ s = \mathit{Nothing} & \mathsf{get}\ (\mathsf{effect}\ f\ g)\ s = \mathsf{get}\ g\ s \\
\mathsf{get}\ (\mathsf{addfst}\ f)\ (s_1, s_2) = \mathit{Just}\ s_2 & \mathsf{get}\ (\mathsf{addsnd}\ f)\ (s_1, s_2) = \mathit{Just}\ s_1 \\
\mathsf{get}\ (\mathsf{remfst}\ f)\ s\ = \mathit{Just}\ (f\ s, s) & \mathsf{get}\ (\mathsf{inj}\ p)\ (\mathit{Left}\ s)\ \ = \mathit{Just}\ s \\
\mathsf{get}\ (\mathsf{remsnd}\ f)\ s = \mathit{Just}\ (s, f\ s) & \mathsf{get}\ (\mathsf{inj}\ p)\ (\mathit{Right}\ s) = \mathit{Just}\ s \\
\multicolumn{2}{l}{\mathsf{get}\ (f \otimes g)\ (s_1, s_2) = \mathbf{do}} \\
\multicolumn{2}{l}{\quad \{ v_1 \leftarrow \mathsf{get}\ f\ s_1; v_2 \leftarrow \mathsf{get}\ g\ s_2; \mathit{return}\ (v_1, v_2) \}} \\
\multicolumn{2}{l}{\mathsf{get}\ (f \mathbin{\nabla} g)\ s \mid \mathit{isNothing}\ (\mathsf{get}\ f\ s) = \mathit{fmap}\ \mathit{Right}\ (\mathsf{get}\ g\ s)} \\
\multicolumn{2}{l}{\qquad\qquad\quad \mid \mathit{isNothing}\ (\mathsf{get}\ g\ s) = \mathit{fmap}\ \mathit{Left}\ (\mathsf{get}\ f\ s)} \\
\multicolumn{2}{l}{\qquad\qquad\quad \mid \mathbf{otherwise} = \mathit{Nothing}} \\
\multicolumn{2}{l}{\mathsf{get}\ (f \oplus g)\ (\mathit{Left}\ s_1)\ \ = \mathit{fmap}\ \mathit{Left}\ (\mathsf{get}\ f\ s_1)} \\
\multicolumn{2}{l}{\mathsf{get}\ (f \oplus g)\ (\mathit{Right}\ s_2) = \mathit{fmap}\ \mathit{Right}\ (\mathsf{get}\ g\ s_2)} \\
\mathsf{get}\ \mathsf{in}\ s = \mathit{out}\ s & \mathsf{get}\ \mathsf{out}\ s = \mathit{inn}\ s
\end{array}
$$

Figure 1. Explicit get functions for our putlens language.

s, we can know whether $get\ s$ is defined or not given that it returns a view value $Just\ v$ or $Nothing$[13]. Extending get with the $Maybe$ monad is merely an implementation detail, as users don't write get functions. Alternatively to Definition 3.1, we can characterize well-behaved putlenses in terms of put and get functions.

Proposition 3. *A putlens $l :: s \Leftarrow_m v$ is well-behaved iff it satisfies the following properties:*

$$s' \in \mathsf{put}\ s\ v' \;\Rightarrow\; Just\ v' = \mathsf{get}\ s' \qquad \textsc{PutGet}_{\Leftarrow}$$

$$Just\ v \in \mathsf{get}\ s \;\Rightarrow\; \mathsf{put}\ (Just\ s)\ v = return\ s \qquad \textsc{GetPut}_{\Leftarrow}$$

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 $\textsc{PutTwice}_{\Leftarrow}$. A technical remark is that, for some combinators like remfst or Φ, 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 $s = Just\ \bot$); 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 putlens composition ($\circ\!\!<$). 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 putlens framework into an equivalent formulation but where get and put are tupled into a single function, as originally suggested in [27]:

$$
\begin{array}{l}
\mathbf{data}\ s \Leftarrow_m v = Putlens\ \{\mathsf{create} :: v \to m\ s \\
\quad , \mathsf{getput} :: s \to (Maybe\ v, v \to m\ s)\}
\end{array}
$$

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 original source to produce a (maybe) original view and a create function for the case when the original source is available.

6. Related Work

BXs in the database community are known under the classical view update problem [1]. To tackle the ambiguity of view update translation, some approaches [16, 17] propose interactive algorithms that run a dialog with the view programmer and the view user to choose a particular view update strategy obeying a set of intuitive criteria. Unlike put programming, dialog approaches only consider update translations deemed intuitive (to reduce the number of questions asked), but not all well-behaved update translators.

All existing bidirectional programming approaches based on lenses focus on writing bidirectional programs that resemble get functions. Some examples are lens languages or techniques for generalized trees [10], algebraic data types [14, 20, 23, 25, 28], XML data [15, 26, 27], strings [2, 5], relational data [4] or graphs [13].

Some of these approaches provide stronger guarantees by ensuring that all lenses are total [2, 4, 5, 10, 14, 23, 25, 26]. However, meeting totality implies limiting the expressiveness of the supported transformations. (Note that totality and consequent surjectivity are not necessary conditions for the uniqueness of BXs in Theorem 2.1.) For example, the point-free language of [23] does not support injective get functions such as constant functions of type $() \to a$, duplication or injections, all supported as putlenses. Other approaches relax valid programs to partial well-behaved lenses [13, 15, 20, 27, 28], although that may represent a serious drawback on updatability, since users cannot predict when particular view updates will fail simply from the specification of the forward transformation. The seminal lens language of [10] instead assumes a very precise semantic type system to ensure that well-typed lenses are well-behaved and total, despite not providing automated type checking; we follow a similar approach by introducing a set-theoretic type system to reason about totality of well-typed putlenses, but nevertheless ensure that (non-recursive) partial putlenses are well-behaved with respect to the more conventional Haskell type system. In comparison to get-based languages, putlenses may fail for particular inputs but the programmer has a chance of molding the partiality of put (as programming with partial functions in Haskell) by taking into account the semantic annotations, and knowing that (for non-recursive expressions) the combinators map total putlenses to total putlenses.

On a different topic, all the abovementioned bidirectional programming approaches are state-based, meaning that the put function of a lens must align the updated view with the original source to infer a corresponding view update. Although for unordered data (relations, graphs) such alignment can be done rather straightforwardly, for ordered data (strings, trees) it is more problematic to find reasonable alignment strategies. Regardless of this fact, these approaches either consider a single update strategy or provide limited control over the update strategy, and even alignment-aware languages [2, 25] still support restricted user control over specific transformation patterns for particular kinds of data. Operation-based approaches (see [6, 8]), that consider actual update operations, possess more information, but also tend to commit to particular update strategies for particular classes of updates. Conversely, our put-based lens language allows programmers to control the update strategies, making BX behavior more predictable.

The work from [21] proposes a point-free BX language of injective functions and their partial inverses, from which we draw many ideas. But in our terms [21] only amounts to writing injective $create$ functions, while our language of injective $put\ s$ functions supports programming of more challenging put update strategies.

The Haskell `lens` package provides an interface of lenses that define getters/setters for records and other general containers. Lenses in this context are typically projections (in the style of

[13] We define $disjoint\ x\ y\ s = isJust\ (\mathsf{get}\ x\ s) \wedge isNothing\ (\mathsf{get}\ y\ s)$ and $enforceGetPut\ f = Putlens\ put\ (\mathsf{get}\ f)$ with $put\ s\ v = \mathbf{if}\ isJust\ s \wedge \mathsf{get}\ (fromJust\ s) \equiv Just\ v\ \mathbf{then}\ return\ s\ \mathbf{else}\ \mathsf{put}\ f\ s\ v$.

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 putlenses 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 putlenses, 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 putlenses. 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

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[2] D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *ICFP 2010*, pages 193–204. ACM, 2010.

[3] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall, 1997.

[4] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS 2006*, pages 338–347. ACM, 2006.

[5] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL 2008*, pages 407–419. ACM, 2008.

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[7] H. Doornbos and R. Backhouse. Reductivity. *Science of Computer Programming*, 26(1):217 – 236, 1996.

[8] S. Fischer, Z. Hu, and H. Pacheco. "Putback" is the Essence of Bidirectional Programming. Technical Report GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, Dec. 2012.

[9] J. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, December 2009.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[11] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *ICFP 2011*, pages 2–14. ACM, 2011.

[12] M. Hanus (editor). Curry: An integrated functional logic language (vers. 0.8.3), 2012.

[13] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP 2010*, pages 205–216. ACM, 2010.

[14] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *POPL 2011*, pages 371–384. ACM, 2011.

[15] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

[16] A. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB 86*, pages 467–474. Morgan Kaufmann Publishers, 1986.

[17] J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145 – 168, 1991.

[18] N. Macedo, H. Pacheco, A. Cunha, and J. N. Oliveira. Composing least-change lenses. In *Proceeding of the 2nd International Workshop on Bidirectional Transformations (BX 2013)*. to appear, 2013.

[19] S. Marlow (editor). Haskell 2010 language report, 2010.

[20] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP 2007*, pages 47–58. ACM, 2007.

[21] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In *APLAS 2004*, volume 3302 of *LNCS*, 2004.

[22] U. Norell. Dependently Typed Programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.

[23] H. Pacheco and A. Cunha. Generic point-free lenses. In *MPC 2010*, volume 6120 of *LNCS*, pages 331–352. Springer, 2010.

[24] H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *PEPM 2011*, pages 91–100. ACM, 2011.

[25] H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *BX 2012*, volume 49 of *Electronic Comms. of the EASST*, 2012.

[26] R. Rajkumar, S. Lindley, N. Foster, and J. Cheney. Lenses for web data. In *BX 2013*. to appear, 2013.

[27] M. Takeichi. Configuring bidirectional programs with functions. In *IFL 2009*. Seton Hall University, 2009.

[28] J. Voigtländer. Bidirectionalization for free! (pearl). In *POPL 2009*, pages 165–176. ACM, 2009.

[29] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Refactoring pattern matching. *Science of Computer Programming*, 78(11):2216 – 2242, 2013.