

# Contract Lenses: Reasoning about Bidirectional Programs via Calculation

HANLIANG ZHANG

University of Bristol, United Kingdom  
(e-mail: hlzhang1997@pku.edu.cn)

WENHAO TANG

The University of Edinburgh, United Kingdom  
(e-mail: wenhao.tang@ed.ac.uk)

RUIFENG XIE

Peking University, China  
(e-mail: xieruifeng@pku.edu.cn)

MENG WANG

University of Bristol, United Kingdom  
(e-mail: meng.wang@bristol.ac.uk)

ZHENJIANG HU

Peking University, China  
(e-mail: huzj@pku.edu.cn)

---

## Abstract

Bidirectional transformations (BXs) are a mechanism for maintaining consistency between multiple representations of related data. The lens framework, which usually constructs BXs from lens combinators, has become the mainstream approach to BX programming because of its modularity and correctness by construction. However, the involved bidirectional behaviours of lenses make the equational reasoning and optimisation of them much harder than unidirectional programs. We propose a novel approach to deriving efficient lenses from clear specifications via program calculation, a correct-by-construction approach to reasoning about functional programs by algebraic laws.

To support bidirectional program calculation, we propose *contract lenses*, which extend conventional lenses with a pair of predicates to enable safe and modular composition of partial lenses. We define several contract-lens combinators capturing common computation patterns including *fold*, *filter*, *map*, and *scan*, and develop several bidirectional calculation laws to reason about and optimise contract lenses. We demonstrate the effectiveness of our new calculation framework based on contract lenses with non-trivial examples.

---

## 1 Introduction

A *bidirectional transformation* (BX) is a pair of mappings between *source* and *view* data objects, one in each direction. When the source is updated, a (*forward*) transformation executes to obtain an updated view. For a variety of reasons, the view may also

be subjected to direct manipulation, requiring a corresponding (*backward*) transformation to keep the source consistent. Much work has gone into this area with applications in databases (Bancilhon and Spyrtatos, 1981; Bohannon et al., 2006; Tran et al., 2020), software model transformation (Stevens, 2008; He and Hu, 2018; Tsigkanos et al., 2020; Stevens, 2020), graph transformation (Hidaka et al., 2010) etc; in particular there has been several language-based approaches that allow transformations in both directions to be programmed together (for example Foster et al. (2007); Voigtländer (2009); Matsuda et al. (2007); Ko et al. (2016)).

The *lens* framework (Foster et al., 2007) is the leading approach to BX programming. A lens consists of a pair of transformations: a *forward* transformation *get* producing a *view* from a *source*, and a *backward* transformation *put* which takes a source and a possibly modified view, and reflects the modifications on the view to the source, producing an updated source. It can be represented as a record using Haskell-like notations as

$$\mathbf{data} \ S \leftrightarrow V = \mathit{Lens} \ \{ \mathit{get} : S \rightarrow V, \mathit{put} : S \rightarrow V \rightarrow S \}$$

The additional argument  $S$  in *put* ensures that a view does not have to contain all the information of the source for backward transformation to be viable.

These two transformations should be *well-behaved* in the sense that they satisfy the following *round-tripping properties*:

$$\begin{array}{ll} \mathit{put} \ s \ (\mathit{get} \ s) = s & \text{GETPUT} \\ \mathit{get} \ (\mathit{put} \ s \ v) = v & \text{PUTGET} \end{array}$$

The **GETPUT** property requires that no-change to the view should be reflected as no-change to the source, while the **PUTGET** property requires that all changes in the view should be completely reflected to the source so that the changed view can be successfully recovered by applying the forward transformation to the updated source.

One main advantage of lenses is their modularity. The lens composition  $\ell_1; \ell_2 : S \leftrightarrow T$  of lenses  $\ell_1 : S \leftrightarrow V$  and  $\ell_2 : V \leftrightarrow T$  is defined as <sup>1</sup>

$$\begin{array}{l} \ell_1; \ell_2 = \mathit{Lens} \ g \ p \\ \mathbf{where} \\ \quad g \quad = \mathit{get}_{\ell_2} \circ \mathit{get}_{\ell_1} \\ \quad p \ s \ t' = \mathit{put}_{\ell_1} \ s \ (\mathit{put}_{\ell_2} \ (\mathit{get}_{\ell_1} \ s) \ t') \end{array}$$

In the forward direction, lens composition is simply a function composition of the two *get* functions. In the backward direction, it will first put the updated  $t'$  back to the intermediate  $v$  produced by  $\mathit{get}_{\ell_1} \ s$  using  $\ell_2$ , and then put the updated  $v$  back to  $s$ .

Lenses are programmed in special languages that preserve round-tripping properties by construction. One popular type of such languages are lens combinators, i.e., higher order functions that construct complex lenses by composing simpler ones. Designing lens languages that are expressive and easy-to-use has been a popular research topic (Bohannon et al., 2008; Foster et al., 2008; Barbosa et al., 2010; Hofmann et al., 2011; Ko et al., 2016; Matsuda and Wang, 2018, 2015), effectively creating the paradigm of bidirectional programming.

<sup>1</sup> Note that the order of the composition of lenses is left-to-right, while the function composition is right-to-left.

93 This flourishing scene of languages invites the next question of software development:  
94 what are the suitable methods of BX program construction?

95 *Is there an algebraic theory of lens combinators that would underpin optimization of lens expres-*  
96 *sions in the same way that the relational algebra and its algebraic theory are used to optimize*  
97 *relational database queries? ... This algebraic theory will play a crucial role in a more serious*  
98 *implementation effort.* (Foster et al., 2007)

99 Motivated by this question, we propose a calculation framework which optimizes lenses  
100 over lists from clear specifications using the algebraic structures of lens combinators.

### 102 *1.1 Program Calculation and the Challenge of Partiality*

104 Program calculation (Bird, 1989) is an established technique for reasoning about and  
105 optimizing functional programs. The idea is that program developments may benefit from  
106 simple properties and laws: equivalences between programming constructs. And conse-  
107 quently, one may *calculate* with programs — in the same way that one calculates with  
108 numeric quantities in algebra — to transform simple specifications into sophisticated and  
109 efficient implementations. Each step of a calculation is a step of equational reasoning,  
110 where properties of a fragment of the program, such as relations between data structures  
111 and algebraic identities, are applied to transform the program structure. A great advantage  
112 of this method is that the resulting implementation is guaranteed to be *semantically equiv-*  
113 *alent* to the original specification, removing the onerous task of verifying the correctness  
114 of the resulting implementation.

115 Our observation is that program calculation is a good fit to BX programming in a number  
116 of different ways. In terms of philosophy, both advocate correctness by construction aiming  
117 at significantly reducing the verification and maintenance effort. In terms of representation,  
118 both rely heavily on forming programs using composition and computation patterns: in BX  
119 languages, the computation patterns are typically captured as lens combinators which are  
120 designed to preserve well-behavedness, and in program calculation, the use of computation  
121 patterns allows general algebraic laws such as fusion laws and Horner’s rule (Gibbons,  
122 2002, 2011) to be applied to specific instances without the need of special analysis.

123 However, the more complex setting of BX as compared to unidirectional programs posts  
124 unique challenges to program calculation. First of all, calculating BX cannot be superfi-  
125 cially treated as calculating twice, once in each direction, as the round-tripping properties  
126 bind *get* and *put* closely together, demanding simultaneous reasoning with both. Moreover,  
127 lenses are often *partially* defined, making it hard to reason about the *construction* and *com-*  
128 *positions* of combinators like *map*, *fold*, and *scan*. Semantic preservation amid calculation  
129 is difficult in this case as well (note that a change in the definedness of a function changes  
130 its semantics).

131 In this context, the term *partiality* links to round-tripping properties. A lens is partial  
132 when its *put* component cannot successfully restore consistency for certain inputs, even  
133 if this function is total (Stevens, 2014).<sup>2</sup> This *partiality* can be *inherent*, where the *get*  
134 component is non-surjective; there is no meaningful *put* semantics for values outside the  
135 codomain of *get*. This partiality can also be of *design choices*, as forcing a lens to be total

136  
137 <sup>2</sup> In this paper, we assume all functions are total.  
138

may introduce unwanted complexity. As an example, consider following definition of list mapping as a (high-order) lens which takes a lens  $\ell$  of type  $A \leftrightarrow B$  and return a lens of type  $[A] \leftrightarrow [B]$ .

$$\begin{aligned} bmap & : (A \leftrightarrow B) \rightarrow ([A] \leftrightarrow [B]) \\ bmap \ell & = Lens (map get_\ell) p \\ \text{where } p & (x : xs) (y : ys) = put_\ell x y : p xs ys \\ p - & - = [] \end{aligned}$$

This lens is partial: when the view list is updated to be longer, the *put* component cannot deal with the inconsistency of the structure (length) between the original source list and the updated view list correctly; it only returns a new source list of the same length as the original one. As a result, the **PUTGET** property is broken, as shown by  $get_{bmap \text{ bid}} (put_{bmap \text{ bid}} [1] [2, 3]) = [2] \neq [2, 3]$  where *bid* is the trivial identity lens. It is common in practice to assume that only certain view updates are permitted, for example, the length of the view list is preserved. With such an assumption, *bmap* serves as a correct lens.

As a remark, for some lenses such as *bmap*, it is possible to make their definitions total without contracts and any other constraints on sources and views by using more complicated machinery such as default values (Foster et al., 2007; Pacheco and Cunha, 2011). However, giving total definitions to lenses (especially their *put* components) requires more involved types and semantics and leads to extra programming work for designing lenses. It is totally not necessary to endure this extra complication when we can guarantee that the changes on views always satisfy certain constraints, such as preserving the structures (lengths) of views. Moreover, forcing total definitions also results in challenges to the development of calculation laws, again due to the additional complications of types and semantics. For example, the calculation law of *bmap* with default values will require additional semantic conditions on them as shown in [Appendix 1](#).

In this work, instead of insisting on giving total definitions to all lenses, we use a pair of predicates to constrain the changes on the source and view, so that partial lenses can be constructed correctly and composed well. It also facilitates the development of simple but powerful calculation laws.

## 1.2 Contributions

In this paper, we develop a calculation framework to reason about and optimize bidirectional programs over lists. Our goal is to transform lenses with clear specifications to efficient ones by applying calculation laws. Specifically, we propose an extension to traditional lenses, which we call *contract lenses*, to enable the construction and composition of possibly partial lenses. We develop several contract-lens combinators, which are high-order functions that characterize key bidirectional computation patterns on lists. And we establish related calculation laws that lay the foundation of a general algebraic theory for BX calculation.

**Contract Lenses** The main idea of contract lenses is to utilize a pair of fine-grained predicates, one on source and one on view, to characterize the bidirectional behaviour on

185 propagating changes in a compositional way. Composition of contract lenses is justified by  
 186 the implication relation between the view predicate of the former lens and the source pred-  
 187 icate of the latter lens. We also provide an equivalence relation between contract lenses for  
 188 calculation. (Section 4)

189 **Contract-Lens Combinators** We develop bidirectional computation patterns on the list  
 190 data structure using *contract lenses*, including bidirectional *fold*, *map* and *scan*. An inter-  
 191 esting finding is that some bidirectional versions of *map* and *scan* cannot be expressed as  
 192 instances of bidirectional *fold* due to the requirement of maintaining the consistency of  
 193 inner dependencies of data structures. (Section 5)

194 **Contract-Lens Calculation Laws** We establish calculation laws that transform compo-  
 195 sitions of such combinators into equivalent but efficient forms. We provide bidirectional  
 196 versions of many algebraic laws, including *fold fusion*, *map fusion*, *fold-map fusion*, and  
 197 the *scan lemma*. These laws comprise a bidirectional algebraic theory that manipulates  
 198 lenses directly, which underpins the optimization of bidirectional programs. (Section 6)

199 **Mechanized Proofs in Agda** We prove the technical details of our calculation framework  
 200 in Agda, including the correctness of all contract lens combinators and calculation laws, as  
 201 well as most of the examples. The proof consists of 4k lines of Agda code. (Section 9 and  
 202 Supplementary Files)

203 Moreover, we showcase the ability of our framework to construct and calculate lenses by  
 204 advanced examples that either have intricate partial bidirectional behaviours, or are well-  
 205 studied in both bidirectional transformations and program calculation literature (Section 7).  
 206 Section 8 discusses related works, and Section 10 concludes.

207 One thing worth noting is that our primary goal is to propose a calculation framework  
 208 without restricting to any specific reasoning method. Users are free to calculate contract  
 209 lenses with pencil/paper proofs following the tradition of program calculation (Bird, 1989),  
 210 or formalise the calculation via theorem provers like our mechanized proofs in Agda. It is  
 211 even possible to develop automatic reasoning tools based on our framework.

## 212 2 Background: Program Calculation

213 Program calculation (Bird, 1989; Gibbons, 2002) is a technique for constructing efficient  
 214 programs that are correct by construction. It is suitable for humans to derive efficient pro-  
 215 grams by hand (Bird, 1989), as well as for compilers to optimize programs automatically  
 216 (Gill et al., 1993; Hu et al., 1996). The principle of program calculation is to express the  
 217 initial specification of the programming problem in terms of a set of higher order functions,  
 218 which support generic algebraic laws, so that an efficient implementation can be calculated  
 219 through a process of equational reasoning based on the algebraic laws.

### 220 2.1 Specification with Folds

221 Fold is a computation pattern that captures structural recursion. In Haskell, there are two  
 222 versions of fold on list:  $foldl : (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$  and  $foldr : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow$   
 223

$[a] \rightarrow b$ , which can be used to define a range of functions. We give some examples as follows, which are also used in the remainder of the paper.

```

231     maximum = foldr max (-∞)
232     sum      = foldr (+) 0
233     map f    = foldr (λ a r → f a : r) []
234     filter p = foldr (λ a r → if p a then a : r else r) []
235     scanr f b0 = foldr (λ a bs → (f a (head bs)) : bs) [b0]
236     inits    = foldr (λ a r → [] : map (a:) r) [[]]
237     tails    = foldr (λ a r → (a : head r) : r) [[]]

```

Here, *maximum* computes the maximum of a list, *sum* sums up all the elements in a list, *map f* applies function *f* to each element of a list, *filter p* accepts a list and keeps those elements that satisfy *p*, *scanr* keeps the intermediate results of *foldr* in a list (similarly we have a *scanl*), *inits* returns all initial segments (prefix lists) of a list, and *tails* returns all tail segments (postfix lists) of a list.

Note that *foldr f e* has two arguments, which can be combined into one *foldr' alg* where *alg* is a function of type *Either () (a, b) → b*.

```

245     foldr' : (Either () (a, b) → b) → [a] → b
246     foldr' alg [] = alg (Left ())
247     foldr' alg (x : xs) = alg (Right (x, foldr' alg xs))

```

Now we have *foldr f e = foldr' alg*, where *alg* is defined below.

```

251     alg (Left ()) = e
252     alg (Right (a, b)) = f a b

```

One advantage of writing *foldr'* this way is that it can be generalized to arbitrary algebraic data types such as trees (Gibbons, 2002), and its dual *unfoldr'* can be easily defined.

```

253     unfoldr' : (b → Either () (a, b)) → b → [a]
254     unfoldr' coalg b = case coalg b of
255         Left () → []
256         Right (a, b) → a : unfoldr' coalg b

```

There are some variants of the above functions that will be used later:

```

261     inits' = tail ∘ inits
262     tails' = init ∘ tails
263     scanl' f x = tail ∘ scanl (flip f) x
264     scanr' f x = init ∘ scanr f x

```

The main difference is that they remove the empty list from the result. For example, *inits'*  $[1, 2, 3] = [[1], [1, 2], [1, 2, 3]]$ .

Note that the functions defined with *fold* are all executable programs. But we call them specifications in the context of program calculation because such definitions (despite being clear and concise) are not necessarily efficient (especially when multiple *fold*s are

composed together). Program calculation is about turning such specifications into more efficient (though likely less clear) implementations.

## 2.2 Algebraic Laws

The foundation of program calculation is the algebraic laws, which can be applied step by step to derive efficient implementations. The most important algebraic law for fold is the *foldr fusion* law:

$$\frac{h \circ f = g \circ F_L h}{h \circ \text{foldr}' f = \text{foldr}' g} \quad \text{FOLD FUSION}$$

It states that a function  $h$  composed with a *foldr'* can be fused into a single *foldr'* if the fusible condition  $h \circ f = g \circ F_L h$  is satisfied. Note that  $F_L$  is the so-called *list functor*, which is defined by

$$F_L h = \text{const } () + id \times h$$

where  $+$  and  $\times$  on functions are defined by  $(f + g)(\text{Left } x) = \text{Left } (f x)$ ,  $(f + g)(\text{Right } y) = \text{Right } (g y)$ , and  $(f \times g)(x, y) = (f x, g y)$ . The function *const* and *id* are defined by  $\text{const } x \_ = x$  and  $id x = x$ .

There is a corresponding fusion law for *foldl* too. And for some special cases of fold, the fusible conditions are always satisfied and therefore omitted from the laws.

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g) \quad \text{MAP FUSION}$$

$$\text{foldr}' f \circ \text{map } g = \text{foldr}' (f \circ F_m g) \quad \text{FOLD-MAP FUSION}$$

$$\text{map } (\text{foldl } f e) \circ \text{inits} = \text{scanl } f e \quad \text{SCAN LEMMA}$$

Note that  $F_m$  is the so-called *map functor*, which is defined by

$$F_m h = \text{const } () + h \times id$$

It is worth noting that it is possible for an algebraic law to abstract a complex derivation step. For instance, the following Horner's lemma shows a big step to fuse a complex composition into a single *foldl*.

**Lemma 1** (Horner's Rule). Let  $\oplus$  and  $\otimes$  are associative operators. Suppose  $\otimes$  distributes through  $\oplus$  and  $b$  is a left-identity of  $\oplus$ , then:

$$\text{foldl } (\oplus) b \circ \text{map } (\text{foldl } (\otimes) a) \circ \text{tails} = \text{foldl } (\odot) a$$

where  $x \odot y = (x \otimes y) \oplus a$ , and  $a$  is the value passed to *foldl* ( $\otimes$ ).  $\square$

## 2.3 A Calculational Example

The maximum segment sum problem (*mss* for short) is to compute the maximum of the sums of the segments in a list. Developing an efficient implementation of it is challenging, and it has become a classic example to show off the power of program calculation.

The idea is to start with a straightforward specification as follows.

$$mss = \text{maximum} \circ \text{map } \text{maximum} \circ \text{map } (\text{map } \text{sum}) \circ \text{map } \text{tails} \circ \text{inits}$$

Given a list, we first enumerate all the segments by  $\text{map tails} \circ \text{inits}$ . Then we calculate the sum of all segments by  $\text{map}(\text{map sum})$  and get the maximum of these results of sum by  $\text{maximum} \circ \text{map maximum}$ . This implementation is easy to understand but very inefficient ( $O(n^3)$  where  $n$  is the length of the list). Through program calculation, one can step-by-step rewrite the program through applying a sequence of algebraic laws to reach a version that has time complexity  $O(n)$ . The details of the calculation can be found in Bird (1989).

The challenge that this paper aims to address is: *Can the same be done for bidirectional programs – deriving efficient lenses from clear specifications?*

### 3 Overview

In this section, we informally introduce contract lenses and demonstrate how they facilitate the construction of a bidirectional program calculation framework.

#### 3.1 Taming Partiality with Contract Lenses

The core idea of contract lenses is to enrich traditional lenses with *source and view conditions* (also called contracts) restricting the changes on source and view, as below

$$\{cs\} \ell \{cv\}$$

where  $\ell$  is a lens with only *get* and *put*. The contracts are highlighted through the paper. Though we write *cs* and *cv* around the lens  $\ell$  for readability in this section, a contract lens is formally defined as a four-tuple consisting of *get*, *put*, *cs*, and *cv*.<sup>3</sup>

This is a BX setting, so we assume that it is the views that are actively updated and the sources are passively changed accordingly. Given a source  $s$  and an updated view  $v$ , the *view condition*  $cv$  is a predicate that takes two arguments: the original view  $\text{get}_\ell s$  and updated view  $v$ , restricting the permitted values of the updated view in relation to the original view. The *source condition*  $cs$  has a similar structure. It takes two arguments: the original source  $s$  and the updated source  $\text{put}_\ell s v$ , specifying an invariant that must hold for source changes as a result of valid view changes.

For the list mapping lens *bmap* we have seen in the introduction, we are interested in a condition that rules out any changes to the structure (length) of the view, which we specify as the following predicate:

$$eqlength = \lambda xs xs' \rightarrow \text{length } xs = \text{length } xs'$$

This condition is enough to ensure that the *put* component of *bmap* can always restore consistency between the updated view and source without breaking the round-tripping properties. In addition, we can conclude for *bmap* that if the view length does not change, the source length does not change either. This gives rise to the following contract lens where  $\{eqlength\}$  serves as both the view and source conditions:<sup>4</sup>

<sup>3</sup> In spite of the similarity of syntax, contracts are different from Hoare logic, which we will discuss in details in [Section 8.2](#).

<sup>4</sup> One might expect a bidirectional version of *map* to have more complicated source and view conditions, e.g., imposing the source and view conditions of the parameter  $\ell$  to all elements in the list. In [Section 5.2](#), we will show alternative definitions of bidirectional map with different contracts.



$\{eqlength\} \text{ bmap } \ell \{eqlength\}$

Two lenses can be composed if the view condition of the former matches the source condition of the latter. For example, we can compose two *bmaps*:

$\{eqlength\} \text{ bmap } \ell_1 \{eqlength\}; \{eqlength\} \text{ bmap } \ell_2 \{eqlength\}$

With contract lenses, the partiality issues of lens composition is reduced to local reasoning of adjacent conditions. Moreover, since we always want the modification on view (and source) to satisfy the contracts, the round-tripping properties also only need to hold when the contracts are satisfied, which significantly simplify the design of lenses. For instance, when designing  $\{eqlength\} \text{ bmap } \ell \{eqlength\}$ , we do not need to consider how to put back the changes to source when the length of view is changed any more.

The idea of introducing contracts is natural because when updating a view of type  $V$  in a BX setting, we usually want the updated view to satisfy certain constraints (like being of the same length as the original view), instead of allowing it to be any value of type  $V$ . Another option of solving the partiality problem is to give total definitions to all lenses. However, as we have discussed in [Section 1.1](#), it leads to several obstacles to designing lenses and developing a calculation framework, which we avoid by using contract lenses.

### 3.2 Calculation with Contract Lenses

Once we have established the composition of contract lenses, we can start to design a calculation framework for lenses.

For the sake of demonstration, we start with a contrived example: given a list of nonempty lists, we extract all head elements of the lists, and then filter out the even elements. (More realistic examples will be given in [Section 7](#).) In the unidirectional setting, one can apply the [FOLD-MAP FUSION](#) law to fuse the two passes of the list as follows:<sup>5</sup>

$$\begin{aligned} & \text{filter even} \circ \text{map head} \\ = & \{ \text{expressing filter as foldr} \} \\ & \text{foldr } (\lambda a r \rightarrow \text{if even } a \text{ then } a : r \text{ else } r) [] \circ \text{map head} \\ = & \{ \text{FOLD-MAP FUSION} \} \\ & \text{foldr } ((\lambda a r \rightarrow \text{if even } a \text{ then } a : r \text{ else } r) \circ \text{head}) [] \end{aligned}$$

With contract-lens combinators, we can give a bidirectional version of the specification.

$\{eqlength\} \text{ bmap } \text{bhead} \{eqlength\}; \{eqlength\} \text{ bfilter even } \{ceven\}$   
**where**  $\text{bhead} = \text{CLens head } (\lambda xs \ x' \rightarrow x' : \text{tail } xs)$

The view condition of *bfilter even* is defined as

$$\text{ceven} = \lambda xs \ x' \rightarrow eqlength \ xs \ x' \wedge \text{all even } x'$$

which depends on the predicate *even*.<sup>6</sup> The combinator *bfilter* is a bidirectional version of *filter* implemented by *bfoldr'*, which is a bidirectional version of *foldr* with contracts

<sup>5</sup> Since all functions are total, here we assume the *head* and *tail* functions only take non-empty lists (for instance, the *List+* type in Agda stdlib implemented as a record of an element and a normal list).

<sup>6</sup> The definition of the contracts of *bfilter* is technically given by the definition of *bfilter*, which will be more clear in [Section 5.1.2](#). Again, we write the contracts around *bfilter* for readability in this section.

(Section 5). We have already seen  $bmap$  in Section 1.1. In this example,  $bfilter\ even$  is also given source and view conditions including  $eqlength$ , which is needed to be composed with  $\{eqlength\} bmap\ bhead\ \{eqlength\}$ . The contracts of  $\{eqlength\} bfilter\ even\ \{ceven\}$  make sense: if the number of even elements is not changed, the total number of elements will neither be changed because the odd elements, which do not appear in the view, remain invariant.

The advantage of calculating with contract lenses is that we only need to care about the round-tripping properties under the source and view conditions, which simplifies the design of lenses, and as a result simplifies the calculation laws. For  $bfoldr'$ , we have a bidirectional version of FOLD-MAP FUSION law called BFOLDER'-BMAP FUSION, with which we can bidirectionalize the calculation process of  $filter\ even \circ map\ head$  we have seen before.<sup>7</sup>

$$\begin{aligned}
& \{eqlength\} bmap\ bhead\ \{eqlength\}; \{eqlength\} bfilter\ even\ \{ceven\} \\
= & \{ \text{expressing } bfilter \text{ as } bfoldr' \} \\
& \{eqlength\} bmap\ bhead\ \{eqlength\}; \{eqlength\} bfoldr'\ (bfilterAlg\ even)\ \{ceven\} \\
= & \{ \text{BFOLDER'-BMAP FUSION} \} \\
& \{eqlength\} bfoldr'\ (bmapF\ bhead; (bfilterAlg\ even))\ \{ceven\}
\end{aligned}$$

The  $bmapF$  is a bidirectional version of  $F_m$  used in the FOLD-MAP FUSION law, and the  $bfilterAlg\ even$  is a bidirectional version of  $\lambda a\ r \rightarrow \mathbf{if}\ even\ a\ \mathbf{then}\ a : r\ \mathbf{else}\ r$  defined in Section 5.1.2.

This “banality” of the calculation is the strength of our framework, as we have successfully set up a system that allows programmers to reason about lenses in almost exactly the same way as they have done for unidirectional programs for decades. In the rest of the paper we will formally develop the contract lens framework and continue to demonstrate the kind of reasoning that it enables through examples far more advanced than the ones we have seen in this section.

## 4 Contract Lenses

In this section we formally define *contract lenses*, a natural extension of the traditional lenses with contracts. This novel construction enables us to express a wide class of partial BXs while ensuring safe and modular composition.

### 4.1 Contract Lenses

Lenses essentially manipulate *changes*. A *put* propagates a change in view back to a change in source with respect to a *get* function. As we have already seen in Section 3, to guarantee correct change propagation, we extend lenses with a pair of constraints,  $cs$  and  $cv$ , describing the conditions of changes in the source and the view respectively.

<sup>7</sup> We omit administrative parameters for contracts taken by higher-order contract lenses  $bfoldr'$  and  $bmapF$  for simplicity. They are easy to be reconstructed from the definitions of lens combinators.

**Definition 1** (Contract Lenses). A contract lens<sup>8</sup> between source of type  $S$  and view of type  $V$  consists of a pair of transformations  $get$  and  $put$  together with a pair of relations: a source condition  $cs : S \rightarrow S \rightarrow Set$  and a view condition  $cv : V \rightarrow V \rightarrow Set$ .

```

461
462
463
464   data S ↔ V = CLens {
465     get : S → V,
466     put : S → V → S,
467     cs  : S → S → Set,
468     cv  : V → V → Set
469   }

```

where the following round-tripping properties are satisfied for every  $s : S$  and  $v : V$ .

$cv (get\ s)\ v \Rightarrow cs\ s\ (put\ s\ v)$	BACKWARDVALIDITY
$cv (get\ s)\ v \Rightarrow get\ (put\ s\ v) = v$	CONDITIONEDPUTGET
$cs\ s\ s \Rightarrow cv (get\ s)\ (get\ s)$	FORWARDVALIDITY
$cs\ s\ s \Rightarrow put\ s\ (get\ s) = s$	CONDITIONEDGETPUT

□

For backward transformations, the **BACKWARDVALIDITY** law and the **CONDITIONEDPUTGET** say that if the change in the view satisfies  $cv$ , then the change in the source should satisfy  $cs$ , and the put-get law holds. For forward transformations, the **FORWARDVALIDITY** law and the **CONDITIONEDGETPUT** say that if the source  $s$  satisfies  $cs\ s\ s$ , then the view  $get\ s$  should satisfy  $cv$ , and the get-put law holds. The condition  $cs\ s\ s$  in the **CONDITIONEDGETPUT** law is necessary to keep the system consistent: if the get-put law  $put\ s\ (get\ s) = s$  holds, replacing  $v$  with  $get\ s$  in the **BACKWARDVALIDITY** law, we have  $cs\ s\ (put\ s\ (get\ s)) = cs\ s\ s$ . The **BACKWARDVALIDITY** law and **FORWARDVALIDITY** law are important for the proof of the Theorem 1, which states that the composition of contract lenses preserves round-tripping properties. Essentially, they guarantee that the contracts are propagated by  $get$  and  $put$ .

We have a few remarks to make here.

First, as we have discussed in Section 1, all functions including  $get$  and  $put$  components of lenses are total in this paper. For simplicity, some function definitions are abridged and lack some catch-all patterns. Complete definitions of these functions can be found in the Agda formalisation.

Second, to be more consistent with our Agda formalisation, we use the  $Set$  type in Agda to represent the type for predicates. Note that any value  $b$  of type  $Bool$  can be transformed into  $Set$  by using the expression  $b = True$ . For readability, we allow this transformation to be implicit in the rest of the paper. That is to say, anywhere a value of type  $Set$  is needed, we can fill in a value of type  $Bool$ .

Third, the role of source conditions in contract lenses are primarily for describing the “effect” on source updates after ruling out those view updates, which can be seen

<sup>8</sup> The name *contract lenses* is inspired by the paradigm of *Programming by Contract*, which requires every function to have a precondition and a postcondition. They are required to hold before entering the function and after leaving the function, respectively.

in the rule **BACKWARDVALIDITY**: when inputs are restricted to satisfy the view condition, the corresponding outputs are *guaranteed* to satisfy the source condition. This guarantee is necessary for contract lens composition. The rule **FORWARDVALIDITY** and **CONDITIONEDGETPUT** are conditioned on  $cs\ s\ s$ , a predicate on the *identity* source update, which should hold in most of the cases. The requirement here is necessary for proving the correctness of contract lens composition. Also note that even though we add conditions to the traditional **GETPUT** and **PUTGET** laws, we do not weaken the properties of lenses. Since we always want them to hold, the condition  $cv\ (get\ s)\ v$  should always be satisfied when we compute  $put\ s\ v$ , and the condition  $cs\ s\ s$  should always be satisfied when we compute  $get\ s$ .<sup>9</sup>

We use the following notational conventions:

- We use  $cs_\ell, cv_\ell, get_\ell, put_\ell$  to refer to the source condition, view condition, forward transformation and backward transformation of a contract lens  $\ell$ , respectively.
- Lists start from index 1 and the notation  $x_i$  refers to the  $i$ -th element of a list  $x$ .

Now we give some simple examples of contract lenses. We leave more interesting examples in [Section 5](#).

**Example 1** (Embedding Traditional Lenses into Contract Lenses). *As contract lenses are extensions of traditional lenses, traditional lenses can be embedded into contract lenses by adding dummy conditions  $ctrue$ , where  $ctrue\ \_ = \top$ .*  $\square$

**Example 2** (Bidirectional Inits). *An interesting example is a bidirectional version of  $inits'$  defined in [Section 2.1](#). The view condition essentially describes the range of the  $inits'$ . It is a little complicated, but this kind of detailed specification is needed for calculation.*

$$\begin{aligned}
 &binits : [a] \leftrightarrow [[a]] \\
 &binits = CLens\ inits'\ p\ eqlength\ cv' \\
 &\quad \text{where } p\ \_ \ v' = \text{if null } v' \text{ then } [] \text{ else last } v' \\
 &\quad \quad cv'\ v\ v' = (\forall 1 < i \leq |v'|, init\ v'_i = v'_{i-1}) \wedge (init\ v'_1 = []) \wedge eqlength\ v\ v'
 \end{aligned}$$

*With the help of the condition on the view change (which keeps the “inits” structure), our putback function becomes very simple, just returning the last element if it is not empty.*  $\square$

## 4.2 Composition of Contract Lenses

Contract lenses are compositional, which is similar to that of traditional lenses, except that we need to be sure that the change conditions match well.

**Definition 2** (Composition of Contract Lenses). *For two contract lenses  $\ell_1 : S \leftrightarrow V$  and  $\ell_2 : V \leftrightarrow T$ , if  $\forall (v : V) (v' : V), cs_{\ell_2}\ v\ v' \Rightarrow cv_{\ell_1}\ v\ v'$  and  $\forall v : V, cv_{\ell_1}\ v\ v \Rightarrow cs_{\ell_2}\ v\ v$  hold, then they can be composed into a contract lens  $\ell_1; \ell_2 : S \leftrightarrow T$  as defined below.*

$$\begin{aligned}
 &\ell_1; \ell_2 = CLens\ g\ p\ cs_{\ell_1}\ cv_{\ell_2} \\
 &\quad \text{where}
 \end{aligned}$$

<sup>9</sup> We write  $\overline{get}$  when we just want to use  $get$  as a total function without considering the satisfaction of the source condition.

$$\begin{aligned}
g &= get_{\ell_2} \circ get_{\ell_1} \\
p s t &= put_{\ell_1} s (put_{\ell_2} (get_{\ell_1} s) t)
\end{aligned}
\quad \square$$

**Theorem 1** (Well-behaved Composition). *For any two contract lenses  $\ell_1 : S \leftrightarrow V$  and  $\ell_2 : V \leftrightarrow T$ , their composition  $\ell_1 ; \ell_2 : S \leftrightarrow T$  satisfies the round-tripping properties.*  $\square$

Notice that we not only require the backward implication  $cs_{\ell_2} v v' \Rightarrow cv_{\ell_1} v v'$ , but also the forward one  $cv_{\ell_1} v v \Rightarrow cs_{\ell_2} v v$ . Intuitively, the latter is used to establish a connection between the **FORWARDVALIDITY** law of  $\ell_1$  and  $\ell_2$ . Moreover, we can strengthen the condition of composition to make it easier to use. We say that two predicates  $c_1 : A \rightarrow A \rightarrow Set$  and  $c_2 : B \rightarrow B \rightarrow Set$  are equivalent, written as  $c_1 \Leftrightarrow c_2$ , if  $A = B$  and  $\forall (a : A) (a' : A), c_1 a a' \Leftrightarrow c_2 a a'$ . The condition of composition can be strengthened to  $cs_{\ell_2} \Leftrightarrow cv_{\ell_1}$ , which is sufficient in most cases.

### 4.3 Equivalence of Contract Lenses

Now we define an equivalence relation over contract lenses.

**Definition 3** (Lens Equivalence). *For lens  $\ell_1 : S \leftrightarrow V$  and  $\ell_2 : S \leftrightarrow V$ , we say  $\ell_1$  is equivalent to  $\ell_2$ , written as  $\ell_1 = \ell_2$ , if*

- $cs_{\ell_1} \Leftrightarrow cs_{\ell_2}$
- $cv_{\ell_1} \Leftrightarrow cv_{\ell_2}$
- $\forall s : S, get_{\ell_1} s = get_{\ell_2} s$
- $\forall (s : S) (v : V), cv_{\ell_1} (get_{\ell_1} s) v \Rightarrow put_{\ell_1} s v = put_{\ell_2} s v$

$\square$

**Theorem 2** (Lens Equivalence is an Equivalence Relation). *The equivalence relation between contract lenses is reflexive, symmetric and transitive.*  $\square$

There is nothing special about this definition of the equivalence relation. The equivalence relation for contract lenses is the base for our equational program reasoning and plays an important role in developing our program calculation theory of contract lenses.

## 5 Contract-Lens Combinators

Lens combinators have become a popular approach to programming bidirectional transformations because of their modularity and correctness-by-construction. In this section, we define several lens combinators to capture fundamental patterns (higher order functions) for easy construction of complex contract lenses in a compositional manner, as well as to demonstrate the expressiveness and flexibility of our new contract lens framework.

Since bidirectional transformations can be considered as unidirectional forward programs with additional *put* semantics, our idea is to bidirectionalise widely used recursion schemes in (forward) functional programming including *fold*, *map*, *filter*, and *scan*. The main challenge is that these functions are usually not bijective, which requires contracts to

make them total and suitable for calculation. Different contracts will lead to different bidirectional version of the same high-order functions, and are useful for different situations. We will give both total bidirectional versions of these functions, and their variants which have some additional conditions on the source and the view to make them flexible for composing with each other. It will be interesting to see later that although *map* and *scan* can be implemented by *fold*, it turns out to be more useful to implement bidirectional versions of *map* and *scan* individually to attain better control over their contracts and behaviours.

### 5.1 Bidirectional Fold

As we have seen in [Section 2.1](#), *folds* are of vital importance in program calculation. We start with *bfoldr*, a bidirectional version of *foldr'*, with trivial source and view conditions.

$$bfoldr : \{\ell : Either () (S, V) \leftrightarrow V \mid cs_\ell \Leftrightarrow true \wedge cv_\ell \Leftrightarrow true\} \rightarrow ([S] \leftrightarrow V)$$

One challenge for designing higher-order contract lenses is that they usually impose certain constraints to the contracts of their lens parameters. For instance, a trivial bidirectional version of *foldr'* requires the parameter lens to have the trivial contract *true*. To specify such requirements, we use similar syntax to refinement types, which is easily readable and understandable by humans, and is also suitable for pencil/paper proofs. In theorem provers, one could use existential types to express the requirements of contracts like our Agda formalisation.

We introduce the following syntactic sugar to specify the source and view conditions of parameters for higher-order contract lenses:

$$\{cs'\} S \leftrightarrow V \{cv'\} \equiv \{\ell : S \leftrightarrow V \mid cs_\ell \Leftrightarrow cs' \wedge cv_\ell \Leftrightarrow cv'\}$$

The type of *bfoldr* can be simplified to

$$bfoldr : (\{true\} Either () (S, V) \leftrightarrow V \{true\}) \rightarrow ([S] \leftrightarrow V)$$

Given a simple contract lens  $\ell : Either () (S, V) \leftrightarrow V$  with trivial contracts, *bfoldr*  $\ell$  returns a contract lens of type  $[S] \leftrightarrow V$  also with trivial contracts, synchronizing a list of type  $[S]$  with a value of type  $V$ . For the *get* direction, we simply use the unidirectional *foldr'*. For the *put* direction, we recursively construct an updated source list (using *unfoldr'*) from the original source and an updated view step by step through *put<sub>ℓ</sub>*, the backward transformation of  $\ell$ . Formally, we define *bfoldr* as follows.

$$bfoldr \ell = CLens (foldr' get_\ell) (curry \$ unfoldr' coalg) true true$$

where

$$\begin{aligned} coalg ([], v') &= \mathbf{case} \text{put}_\ell (\text{Left } ()) v' \mathbf{of} \\ &\quad \text{Left } () \rightarrow \text{Left } () \\ &\quad \text{Right } (a', b') \rightarrow \text{Right } (a', ([], b')) \\ coalg (a : as, v') &= \mathbf{case} \text{put}_\ell (\text{Right } (a, g as)) v' \mathbf{of} \\ &\quad \text{Left } () \rightarrow \text{Left } () \\ &\quad \text{Right } (a', b') \rightarrow \text{Right } (a', (as, b')) \end{aligned}$$

Note that the *put* direction of the above definition is inefficient since it computes “*g as*” every time *coalg*  $(a : as, v')$  is called. A more efficient implementation is to calculate all

645  $g$  as in advance using a *scanr* as shown in Appendix 2.1. We will use the efficient definition  
646 of *bfoldr* in the following sections.

647 Similarly, we can define *bfoldl*, which is omitted here. The following example shows  
648 how the bidirectional fold works.

649 **Example 3** (Bidirectional Maximum). *Considering that we want to synchronize a list with*  
650 *its maximum, we can define it in terms of *bfoldr* by*

651  $bmaximum : [Int] \leftrightarrow Int$   
652  $bmaximum = bfoldr\ bmax$   
653

654 where *bmax* is a bidirectional version of *max* whose backward transformation uses the  
655 modified value to replace the maximum value of the parameter pair.<sup>10</sup>

656  $bmax : Either () (Int, Int) \leftrightarrow Int$   
657  $bmax = CLens\ g\ p\ \mathit{ctrue}\ \mathit{ctrue}$   
658

659 **where**

660  $g\ (\mathit{Left}\ ()) = -\infty$   
661  $g\ (\mathit{Right}\ (x, y)) = \max\ x\ y$   
662  $p\ (\mathit{Left}\ ())\ (-\infty) = \mathit{Left}\ ()$   
663  $p\ (\mathit{Left}\ ())\ v' = \mathit{Right}\ (v', -\infty)$   
664  $p\ (\mathit{Right}\ (x, y))\ v' = \mathbf{if}\ x \geq y\ \mathbf{then}\ \mathit{Right}\ (v', \min\ v'\ y)\ \mathbf{else}\ \mathit{Right}\ (\min\ v'\ x, v')$

665 To see a computation instance of *bmaximum*, we refer to Appendix 3.1. □

### 666 5.1.1 Bidirectional Fold : Preserving Length and Transmitting Constraints

667 While *bfoldr* is useful when it is total in both *get* and *put* directions, we may wish to keep  
668 the length of the source unchanged after *put*. For example, considering the *bmaximum*  
669 in Example 3, we may wish to keep the length of the source list after  $put_{bmaximum}$ , and  
670 furthermore, we hope that the source and view conditions of *bfoldr* be able to express some  
671 extra constraints on the elements. All these can be concisely expressed as the following  
672 higher-order contract lens:  
673

674  $bfoldr' : (\widehat{cs} : S \rightarrow S \rightarrow Set) \rightarrow (\widehat{cv} : V \rightarrow V \rightarrow Set)$   
675  $\rightarrow (\{\mathit{lift}\ \widehat{cs}\ \widehat{cv}\} Either () (S, V) \leftrightarrow V \{\widehat{cv}\}) \rightarrow ([S] \leftrightarrow V)$   
676  $bfoldr' \widehat{cs}\ \widehat{cv}\ \ell = bfoldr\ \ell\ \{cs = \mathit{licond}\ \widehat{cs}, cv = \widehat{cv}\}$   
677

678 The *lift* and *licond*, two high-order predicates, require their arguments to be of the same  
679 shape, and structurally lift predicates over sum types (*Either*) and list types, respectively.

680  $\mathit{lift} : (S \rightarrow S \rightarrow Set) \rightarrow (V \rightarrow V \rightarrow Set) \rightarrow Either () (S, V) \rightarrow Either () (S, V) \rightarrow Set$   
681  $\mathit{lift}\ p\ q\ a\ a' = (a = \mathit{Left}\ () \wedge a' = \mathit{Left}\ ()) \vee$   
682  $(a = \mathit{Right}\ (x, y) \wedge a' = \mathit{Right}\ (x', y') \wedge p\ x\ x' \wedge q\ y\ y')$   
683  $\mathit{licond} : (S \rightarrow S \rightarrow Set) \rightarrow [S] \rightarrow [S] \rightarrow Set$   
684  $\mathit{licond}\ p\ xs\ xs' = \mathit{eqlength}\ xs\ xs' \wedge (\forall 1 \leq i \leq |xs|, p\ xs_i\ xs'_i)$   
685

686 The lens combinator *bfoldr'* takes two predicates  $\widehat{cs}$  and  $\widehat{cv}$  and have the same definition of  
687 *get* and *put* components as *bfoldr*. The  $\widehat{cs}$  represents the constraints on the elements of the  
688

689 <sup>10</sup> We treat  $\infty$  as a value of type *Int* as well for simplicity.

source list, and the  $\widehat{cv}$  represents the view condition. Notice that the predicate parameters  $\widehat{cs}$  and  $\widehat{cv}$  are kind of administrative; their main role is to guarantee that the source condition of the parameter lens is of shape  $\text{lift } cs' \text{ } cv'$  for some  $cs'$  and  $cv'$ . Ideally, we can make them existentially bound. We opt to have explicit predicate parameters to make the presentation clear and more consistent with our Agda formalisation.

**Example 4** (Bidirectional Maximum Preserving Length). *A direct use of  $\text{bfoldr}'$  is to define a bidirectional version of maximum that preserves the length of the source list.*

$$\begin{aligned} b\text{maximum}' &: [\text{Int}] \leftrightarrow \text{Int} \\ b\text{maximum}' &= \text{bfoldr}' \text{ eqlength } \widehat{cv} \text{ } b\text{max}' \\ \text{where} \\ \widehat{cv} &= \lambda x \ x' \rightarrow x \neq -\infty \vee x' = -\infty \\ b\text{max}' &= b\text{max} \{cs = \text{lift } \text{ctrue } cv, cv = \widehat{cv}\} \end{aligned}$$

One may doubt that the  $\text{put } (\text{Left } ()) \ v' = \text{Right } (v', -\infty)$  in  $b\text{max}'$  might break the equal length condition. In fact, it will never be executed because the view condition requires the maximum value to be unchanged when it is  $-\infty$ .  $\square$

### 5.1.2 Bidirectional Filter

As an application of bidirectional folds, we construct the bidirectional filter, which appears frequently in application scenarios of BXs, often in the forms of explicit combinators (Foster et al., 2007) or SQL selection commands (Abou-Saleh et al., 2018).

The unidirectional version of *filter* can be implemented by *foldr* as  $\text{filter } pr = \text{foldr } (\lambda x \ xs \rightarrow \text{if } pr \ x \ \text{then } x : xs \ \text{else } xs) []$ , which returns a list of elements satisfying the predicate  $pr$ . With the  $\text{bfoldr}'$  introduced above, we are able to define a bidirectional version of *filter* which preserves the lengths of the source and view lists.

$$\begin{aligned} b\text{filter} &: (pr : a \rightarrow \text{Bool}) \rightarrow ([a] \leftrightarrow [a]) \\ b\text{filter } pr &= \text{bfoldr}' \ \text{ctrue} \ (f\text{cond } pr) \ (b\text{filterAlg } pr) \\ \text{where} \\ b\text{filterAlg} &: (pr : a \rightarrow \text{Bool}) \rightarrow (\text{Either } () \ (a, [a]) \leftrightarrow [a]) \\ b\text{filterAlg } pr &= \text{CLens } g \ p \ (\text{lift } \text{ctrue} \ (f\text{cond } pr)) \ (f\text{cond } pr) \\ \text{where} \\ g \ (\text{Left } ()) &= [] \\ g \ (\text{Right } (x, xs)) &= \text{if } pr \ x \ \text{then } x : xs \ \text{else } xs \\ p \ (\text{Left } ()) \ [] &= \text{Left } () \\ p \ (\text{Right } (x, xs)) \ xs' &= \text{Right } (\text{if } pr \ x \ \text{then } (\text{head } xs', \text{tail } xs') \ \text{else } (x, xs')) \end{aligned}$$

The function  $f\text{cond}$  is defined as  $f\text{cond } pr = \text{licond } (\lambda \_ \ x' \rightarrow pr \ x')$ . The  $b\text{filterAlg } pr$  is essentially a bidirectional version of the function  $\lambda x \ xs \rightarrow \text{if } pr \ x \ \text{then } x : xs \ \text{else } xs$ . One example of  $b\text{filter}$  is the  $b\text{filter even}$  defined in Section 3.2.

### 5.2 Bidirectional Map

*Map* is another important high-order function in functional programming and program calculation, which applies a function to each element of a list. In this section, we will give



three different definitions of bidirectional *map* with different source and view conditions. The first one is *bmap*, which is just a bidirectional *map* that preserves the length of the source and view list. It has no other constraints on the source and view. The second one is *bmap'*, which takes the constraint on individual elements of the list into consideration. The third one is *bmapl* (and *bmapr*), which goes a step further and takes into account the constraints on adjacent elements of the list as well. These three bidirectional versions of *map* cover a large range of applications. In particular, the most powerful *bmapl* is helpful in our later calculation of bidirectional maximum segment sum.

### 5.2.1 Bidirectional Map: Preserving Length

First, we give *bmap* which preserves the lengths of both source and view lists. It simply requires the parameter to have trivial contracts like *bfoldr*.

$$\begin{aligned} \mathit{bmap} &: (\{\mathit{ctrue}\} S \leftrightarrow V \{\mathit{ctrue}\}) \rightarrow ([S] \leftrightarrow [V]) \\ \mathit{bmap} \ell &= \mathit{CLens} (\mathit{map} \mathit{get}_\ell) p \mathit{eqlength} \mathit{eqlength} \\ &\mathbf{where} \ p \ \mathit{as} \ \mathit{bs}' = \mathit{map} (\lambda(x, y) \rightarrow \mathit{put}_\ell x y) (\mathit{zip} \ \mathit{as} \ \mathit{bs}') \end{aligned}$$

It is clear to see that if the change on the view does not change its length, after backward propagation through  $\mathit{put}_{\mathit{bmap} \ell}$ , the length of the source will not be changed.

As shown in [Section 2.1](#), *map* is just a special version of *fold*. Similarly, we can also implement *bmap* using *bfoldr'* as shown in [Appendix 2.2](#). One example of *bmap* is the *bmap bhead* defined in [Section 3.2](#).

### 5.2.2 Bidirectional Map: Preserving Inner Constraints

The above *bmap* assumes that the lens argument it takes never introduces any constraint. But this is not always the case. When the parameter lens has non-trivial contracts, the bidirectional map combinator should reflect these contracts in its result lens. Thus, we define another version of bidirectional *map* which takes the inner constraints on elements of lists into consideration.

$$\begin{aligned} \mathit{bmap}' &: (S \leftrightarrow V) \rightarrow ([S] \leftrightarrow [V]) \\ \mathit{bmap}' \ell &= \mathit{bmap} \ell \{cs = \mathit{licond} \ \mathit{cs}_\ell, cv = \mathit{licond} \ \mathit{cv}_\ell\} \end{aligned}$$

The *bmap'* simply lifts the contracts of its parameter to all elements in the source and view lists. As seen above, *bmap'* is a generalized version of *bmap*; they are equivalent when the parameter lens  $\ell$  has trivial contracts. Also, we can implement *bmap'* using *bfoldr'* in the same way as shown in [Appendix 2.2](#). One example of *bmap'* is shown in [Appendix 3.2](#)

In the above definition of *bmap'*  $\ell$ , we directly use  $\mathit{cs}_\ell$  and  $\mathit{cv}_\ell$  in the contracts of the result lens. The *bmap'*  $\ell$  has no requirement on the contracts of  $\ell$ . Another alternative definition of *bmap'* more similar to the definition of *bfoldr'* which takes predicate parameters is as follows:

$$\begin{aligned} \mathit{bmap}' &: (\widehat{cs} : S \rightarrow S \rightarrow \mathit{Set}) \rightarrow (\widehat{cv} : V \rightarrow V \rightarrow \mathit{Set}) \\ &\rightarrow (\{\widehat{cs}\} S \leftrightarrow V \{\widehat{cv}\}) \rightarrow ([S] \leftrightarrow [V]) \\ \mathit{bmap}' \ \widehat{cs} \ \widehat{cv} \ \ell &= \mathit{bmap} \ \ell \{cs = \mathit{licond} \ \widehat{cs}, cv = \mathit{licond} \ \widehat{cv}\} \end{aligned}$$

We use the first definition in the paper as it takes fewer arguments.

## 5.2.3 Bidirectional Map: Preserving Constraints on Adjacent Elements

In practice, it is very common that  $mapf$  is composed with a function that produces a list with some constraints on adjacent elements. For instance,  $mapf$  may be composed with  $inits'$ , where the result of  $[as_1, as_2, \dots, as_n]$  produced by  $inits' [a_1, a_2, \dots, a_n]$  has the constraint  $(init\ as_i = as_{i-1}) \wedge (init\ as_1 = [])$ .

In bidirectional programming, we need to carefully specify this kind of constraints. Recall the  $binit$ s in Section 4.1 with the following view condition:

$$cv_{binit} = \lambda t\ as \rightarrow (\forall 1 < i \leq |as|, init\ a_i = a_{i-1}) \wedge (init\ a_1 = []) \wedge eqlength\ t\ as$$

The composition  $binit; bmap\ \ell$  inviolates the condition in Definition 2. This motivated us to introduce  $bmapl$ , another bidirectional version of  $map$  which is able to express constraints on adjacent elements.

The core idea is that for  $bmapl\ \ell$ , we augment the parameter lens  $\ell$  of type  $S \leftrightarrow V$  with an extra argument of type  $S$  representing the adjacent element, which leads to a parameterised lens  $\ell' : S \rightarrow (S \leftrightarrow V)$ . Notice that  $\ell'$  is still a bidirectional version of a function of type  $S \rightarrow V$ , so we need to restrict the  $get$  components of all  $\ell'$ s to be the same function for any  $s : S$ . We again use similar syntax to refinement types to express the requirement on the parameters, and define the following syntactic sugar:

$$A \Rightarrow (S \leftrightarrow V) \equiv \{\ell : A \rightarrow (S \leftrightarrow V) \mid \exists f : S \rightarrow V. \forall a : A. get_{\ell a} = f\}$$

Our two syntactic sugars can be used nestedly:

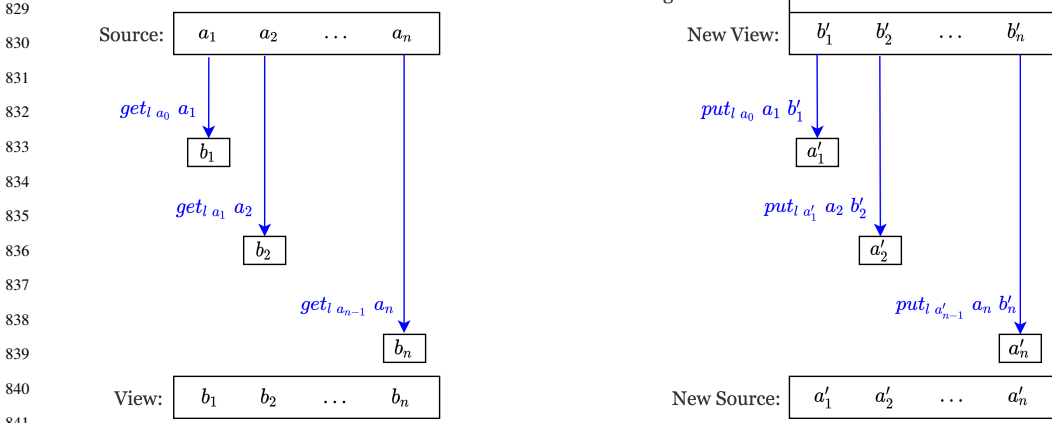
$$A \Rightarrow (\{cs'\} S \leftrightarrow V \{cv'\}) \equiv \{\ell : A \rightarrow \{\ell' : S \leftrightarrow V \mid cs_{\ell} \Leftrightarrow cs' \wedge cv_{\ell} \Leftrightarrow cv'\} \mid \exists f : S \rightarrow V. \forall a : A. get_{\ell a} = f\}$$

The bidirectional  $map$  preserving constraints on adjacent elements is defined as follows:

$$\begin{aligned} bmapl & : (\tilde{cs} : S \rightarrow S \rightarrow Set) \rightarrow (\tilde{cv} : V \rightarrow V \rightarrow Set) \rightarrow (as_0 : S) \\ & \rightarrow (\ell : (a : S) \Rightarrow (\{\lambda_{-} a' \rightarrow \tilde{cs}\ a\ a'\} S \leftrightarrow V \{\lambda_{-} b' \rightarrow \tilde{cv}\ (\overline{get}_{(\ell a)}\ a)\ b'\})) \\ & \rightarrow ([S] \leftrightarrow [V]) \\ bmapl\ \tilde{cs}\ \tilde{cv}\ as_0\ \ell & = CLens\ g\ p\ cs'\ cv' \\ \text{where } bs_0 & = get_{(\ell\ as_0)}\ as_0 \\ g\ as & = map\ (\lambda (a', a) \rightarrow get_{\ell a'}\ a)\ (zip\ (as_0 : init\ as)\ as) \\ p\ as\ bs' & = scanl'\ (\lambda (a, b')\ a' \rightarrow put_{\ell a'}\ a\ b')\ as_0\ (zip\ as\ bs') \\ cs'\ t\ as & = (\forall 1 \leq i \leq |as|. \tilde{cs}\ as_{i-1}\ as_i) \wedge eqlength\ t\ as \\ cv'\ t\ bs & = (\forall 1 \leq i \leq |as|. \tilde{cv}\ bs_{i-1}\ bs_i) \wedge eqlength\ t\ bs \end{aligned}$$

The constraints on adjacent elements of lists are specified by  $\tilde{cs}$  and  $\tilde{cv}$ . For example, if we take  $\tilde{cs}$  to be  $\lambda x\ y \rightarrow (init\ y = x)$  and  $as_0$  to be  $[]$ , then the source condition of the  $bmapl\ as_0\ \ell$  is equivalent to  $cv_{binit}$ , and thus, the composition  $binit; bmapl\ []\ \ell$  is valid.

The implementation of  $bmapl\ a_0\ \ell$  is visualized in Figure 1. The parameterised lens  $\ell : (a : S) \Rightarrow (\{\lambda_{-} a' \rightarrow \tilde{cs}\ a\ a'\} S \leftrightarrow V \{\lambda_{-} b' \rightarrow \tilde{cv}\ (\overline{get}_{(\ell a)}\ a)\ b'\})$  takes the adjacent element of source as the argument. As we have mentioned in Section 4.1,  $\overline{get}_{\ell a}$  means using the  $get$  component of  $\ell\ a$  simply as a total function. For the  $get$  direction, when computing  $b_i$  from  $a_i$ , we pass the adjacent element  $a_{i-1}$  to  $\ell$  and make sure that we have  $\tilde{cv}\ b_{i-1}\ b_i$ , which ensures the view list satisfies the constraints on adjacent elements. For the  $put$  direction,



842 Fig. 1. Implementation of *bmapl*. The left figure shows the computation of the *get* and the right  
 843 figure shows the computation of the *put*.

844 when computing  $a'_i$  from  $b'_i$  and  $a_i$ , we pass  $a'_{i-1}$  to  $\ell$  and make sure that we have  $\tilde{c}s \ a'_{i-1} \ a'_i$ ,  
 845 which ensures the updated source list satisfies the constraints on adjacent elements.

846 Note that we use the name *bmapl* because the constraints are leftwards on every pair of  
 847  $a_{i-1}$  and  $a_i$ . Similarly, we have a *bmapr* which are used to deal with constraints rightwards  
 848 on every pair of  $a_i$  and  $a_{i+1}$ , usually generated by some *scanr'* ( $\oplus$ )  $a_0$ . The implementation  
 849 is almost the same except for replacing *scanl'* in the code with *scanr'*. One example of  
 850 *bmapl* is shown in Appendix 3.3.

#### 852 5.2.4 Bidirectional Map using Inner Bidirectional Fold

853 As we have seen so far, *bmapl*  $\ell$  is useful to give a bidirectional version for *mapf* with  
 854 expressive constraints. What if  $f$  is a *fold*? Since *bmapl* takes a parameterised lens of type  
 855  $S \Rightarrow (S \leftrightarrow V)$ , we cannot directly pass either *bfoldr* or *bfoldr'* to *bmapl*. Moreover, since  
 856 the bidirectional *fold* we needed depends on the  $\tilde{c}s$  in the source condition of the result of  
 857 *bmapl*, it is actually difficult to give a general bidirectional *fold*. Fortunately, we can define  
 858 some special bidirectional versions of *fold* to cope with some frequently used constraints,  
 859 such as  $\lambda a_{i-1} \ a_i \rightarrow \text{init } a_i = a_{i-1}$ . The *bfoldl\_init* shown below is such a special *bfold* that  
 860 can be used inside *bmapl*.

861

862  $bfoldl_{init} : (\tilde{c}v : V \rightarrow V \rightarrow Set) \rightarrow (b_0 : V)$

863  $\rightarrow (\ell : (b : V) \Rightarrow \{\lambda \_ t' \rightarrow t' = \text{Right } (\_, b)\}$

864  $\text{Either } () \ (S, V) \leftrightarrow V$

865  $\{\lambda \_ b' \rightarrow \tilde{c}v \ b \ b'\})$

866  $\rightarrow [S] \Rightarrow ([S] \leftrightarrow V)$

867  $bfoldl_{init} \ \tilde{c}v \ b_0 \ \ell \ as = \text{CLens } g \ p \ cs' \ cv'$

868 **where**  $g = \text{foldl } (\lambda b \ a \rightarrow \text{get}_{\ell} b \ (\text{Right } (a, b))) \ b_0$

869  $p \ [] \ b' = \text{case } \text{put}_{\ell} (g \ as) \ (\text{Left } ()) \ b' \ \text{of}$

870  $\text{Right } (a, \_) \rightarrow as \ \# \ [a]$

871  $p \ as' \ b' = \text{case } \text{put}_{\ell} (g \ as) \ (\text{Right } (\text{last } as', g \ (\text{init } as')) \ b' \ \text{of}$

872  $\text{Right } (a, \_) \rightarrow as \ \# \ [a]$

873

874



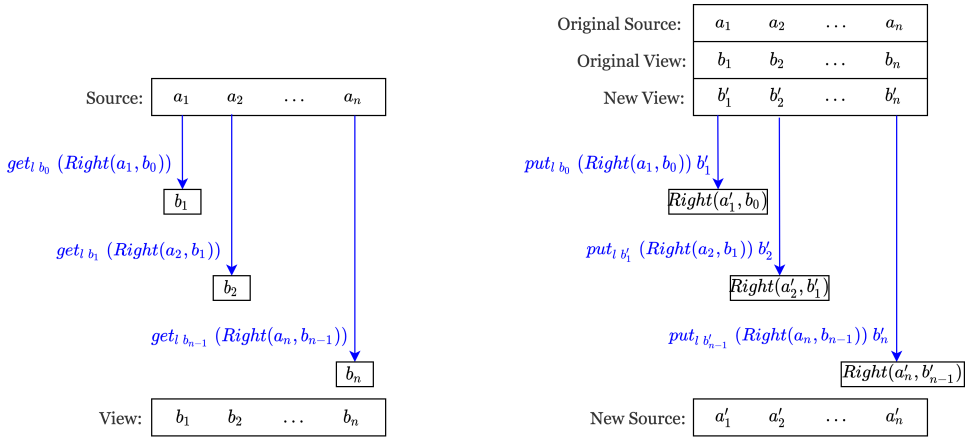


Fig. 2. Implementation of `bscanl`. The left figure shows the computation of the `get` and the right figure shows the computation of the `put`.

several important bidirectional calculation laws for manipulating them, including bidirectional versions of **FOLD FUSION**, **MAP FUSION** and **SCAN LEMMA**. These bidirectional calculation laws are useful to reason about and optimize bidirectional programs.

### 6.1 Bidirectional Fold Fusion

We start with a bidirectional version of the **FOLD FUSION** law for `bfoldr`. To characterize bidirectional fold fusion law, we first bidirectionalize the list functor  $F_L$  in **Section 2.2**.

$$\begin{aligned}
 \text{blistF} &: V \\
 &\rightarrow (\{\text{ctrue}\} V \leftrightarrow T \{\text{ctrue}\}) \\
 &\rightarrow ((\text{Either } () (S, V)) \leftrightarrow (\text{Either } () (S, T))) \\
 \text{blistF } b_0 \ell &= \text{CLens } g \text{ p } \text{ctrue } \text{ctrue} \\
 \text{where } g (\text{Left } ()) &= \text{Left } () \\
 g (\text{Right } (a, b)) &= \text{Right } (a, \text{get}_\ell b) \\
 p - (\text{Left } ()) &= \text{Left } () \\
 p (\text{Right } (a, b)) (\text{Right } (a', c')) &= \text{Right } (a', \text{put}_\ell b c') \\
 p (\text{Left } ()) (\text{Right } (a', c')) &= \text{Right } (a', \text{put}_\ell b_0 c')
 \end{aligned}$$

The tricky part lies in the last line above when there is a mismatch in the constructors of source and view. The implementation chooses a default value  $b_0$  of type  $V$  to help with this process. With this bidirectional list functor, we can have the following *bidirectional fold fusion law*, which is similar to the unidirectional fold fusion law but with this explicit default value.

$$\frac{\ell_1; \ell = \text{blistF } (\text{get}_{\ell_1} (\text{Left } ())) \ell; \ell_2}{\text{bfoldr } \ell_1; \ell = \text{bfoldr } \ell_2} \quad \text{BFOLDER FUSION}$$

It reads that the lens composition `bfoldr`  $\ell_1; \ell$  can be fused into a single lens `bfoldr`  $\ell_2$  if there exists  $\ell_2$  such that the equation  $\ell_1; \ell = \text{blistF } (\text{get}_{\ell_1} (\text{Left } ())) \ell; \ell_2$  holds.

Similarly, we have another fusion law for `bfoldr'`, for which we need a slightly different bidirectional version of the list functor  $F_L$ . The good thing is that we do not need the

default value anymore because the contracts of  $bfoldr'$  guarantee that there will not be any mismatch.

$$\begin{aligned}
967 \quad & \text{blistF}' : (S \rightarrow S \rightarrow \text{Set}) \rightarrow (V \leftrightarrow T) \\
968 \quad & \rightarrow (\text{Either } () (S, V) \leftrightarrow \text{Either } () (S, T)) \\
969 \quad & \text{blistF}' \widehat{cs} \ell = \text{CLens } g \ p \ (\text{lift } \widehat{cs} \ \widehat{cv}) \ (\text{lift } \widehat{cs} \ \widehat{ct}) \\
970 \quad & \text{where } g \ (\text{Left } ()) = \text{Left } () \\
971 \quad & \quad \quad g \ (\text{Right } (a, b)) = \text{Right } (a, \text{get}_\ell b) \\
972 \quad & \quad \quad p \ (\text{Left } ()) \ (\text{Left } ()) = \text{Left } () \\
973 \quad & \quad \quad p \ (\text{Right } (a, b)) \ (\text{Right } (a', c')) = \text{Right } (a', \text{put}_\ell b \ c') \\
974 \quad & \quad \quad \widehat{cv} = \text{cs}_\ell \\
975 \quad & \quad \quad \widehat{ct} = \text{cv}_\ell
\end{aligned}$$

Then, the fusion law is stated as

$$\frac{\ell_1; \ell = \text{blistF}' \widehat{cs} \ell; \ell_2}{\text{bfoldr}' \widehat{cs} \widehat{cv} \ell_1; \ell = \text{bfoldr}' \widehat{cs} \widehat{ct} \ell_2} \quad \text{BFOLDER' FUSION}$$

## 6.2 Bidirectional Map Fusion

The bidirectional map fusion laws for  $bmap$  and  $bmap'$  are quite easy since they just map  $\ell$  to each element of the list in both forward and backward transformations. Since  $bmap$  is a special case of  $bmap'$ , we only give the bidirectional map fusion law for  $bmap'$ .

$$bmap' \ell_1; bmap' \ell_2 = bmap' (\ell_1; \ell_2) \quad \text{BMAP' FUSION}$$

Similarly, we can give the bidirectional map fusion law for  $bmapl$ :

$$bmapl \widehat{cs} \widehat{cv} a_0 \ell_1; bmapl \widehat{cv} \widehat{ct} b_0 \ell_2 = bmapl \widehat{cs} \widehat{ct} a_0 (\ell_1; \ell_2) \quad \text{BMAPL FUSION}$$

where  $(; ;)$  is the composition of parameterised lenses whose types are of form  $S \Rightarrow (S \leftrightarrow V)$ . It is defined as follows:

$$\begin{aligned}
995 \quad & (; ;) : (\ell_1 : (a : S) \Rightarrow (\{\lambda_{-} a' \rightarrow \widehat{cs} a \ a'\} S \leftrightarrow V \ \{\lambda_{-} b' \rightarrow \widehat{cv} (\overline{\text{get}}_{\ell_1 a} a) \ b'\})) \\
996 \quad & \rightarrow (\ell_2 : (b : V) \Rightarrow (\{\lambda_{-} b' \rightarrow \widehat{cv} b \ b'\} V \leftrightarrow T \ \{\lambda_{-} c' \rightarrow \widehat{ct} (\overline{\text{get}}_{\ell_2 b} b) \ c'\})) \\
997 \quad & \rightarrow S \quad \quad \quad \Rightarrow (S \leftrightarrow T) \\
998 \quad & \ell_1; ; \ell_2 = \lambda a \rightarrow \ell_1 a; \ell_2 (\text{get}_{\ell_1 a} a)
\end{aligned}$$

The definition of  $\ell_1; ; \ell_2$  is quite intuitive. We just pass the parameter  $a$  to  $\ell_1$ , and the result of  $a$  after the forward transformation of  $\ell_1$  to  $\ell_2$ . Notice that we still use the syntactic sugar  $S \Rightarrow (S \leftrightarrow T)$  for the type of the result parameterised lenses, which means the  $\text{get}$  component is the same for any parameter. This makes natural sense because both  $\ell_1$  and  $\ell_2$  have fixed  $\text{get}$  components. It is also easy to check that the composition  $\ell_1 a; \ell_2 (\text{get}_{\ell_1 a} a)$  is well-defined (i.e., satisfies the condition in [Definition 2](#)).

## 6.3 Bidirectional Fold-Map Fusion

We give a bidirectional fold-map fusion law for  $bfoldr'$  and  $bmap'$ , both of which preserve the length of the source list.

First, we bidirectionalize  $F_m$  defined in [Section 2.2](#) with conditions required by  $bfoldr'$ .

$$\begin{aligned}
bmapF &: (T \rightarrow T \rightarrow Set) \rightarrow (S \leftrightarrow V) \\
&\rightarrow (Either () (S, T) \leftrightarrow Either () (V, T)) \\
bmapF \widehat{c}t \ell &= CLens g p (lift \widehat{c}s \widehat{c}t) (lift \widehat{c}v \widehat{c}t) \\
\text{where } g (Left ()) &= Left () \\
g (Right (a, c)) &= Right (get_\ell a, c) \\
p (Left ()) (Left ()) &= Left () \\
p (Right (a, c)) (Right (b', c')) &= Right (put_\ell a b', c') \\
\widehat{c}s &= cs_\ell \\
\widehat{c}v &= cv_\ell
\end{aligned}$$

The result of  $bmapF$  has the same source condition as the lens  $bfoldr'$  takes. Now we can give the bidirectional fold-map fusion law for  $bfoldr'$ .

$$bmap' \ell_1; bfoldr' \widehat{c}v \widehat{c}t \ell_2 = bfoldr' \widehat{c}s \widehat{c}t (bmapF \widehat{c}t \ell_1; \ell_2) \quad \text{BFOLDER'-BMAP FUSION}$$

#### 6.4 Bidirectional Scan Lemma

In the unidirectional world, the [SCAN LEMMA](#) is a special version of the [FOLD FUSION](#) law. Note that replacing  $inits$  with  $inits'$  and  $scanl$  with  $scanl'$ , the scan lemma still holds. The major challenge for developing a similar bidirectional calculation law on contract lenses is that the  $inits'$  introduces a constraint on adjacent elements of the view list. Fortunately, the contract-lens combinator  $bmapl$  can handle constraints on adjacent elements. With  $bmapl$ ,  $bfoldl_{init}$  and  $bscanl$ , we can successfully obtain a bidirectional version of scan lemma.

$$binits; bmapl (\lambda a a' \rightarrow init a' = a) \widehat{c}v [] (bfoldl_{init} \widehat{c}v b_0 \ell) = bscanl \widehat{c}v b_0 \ell$$

BIDIRECTIONAL SCAN LEMMA

The form of the bidirectional scan lemma is quite similar to its unidirectional version modulo some administrative parameters for contracts. We give an example of [BIDIRECTIONAL SCAN LEMMA](#) in [Appendix 3.6](#).

## 7 Examples

In this section, we will demonstrate further through three examples that with contract lenses, combinators and associated calculation laws, we are able to flexibly construct and optimize bidirectional programs. The first example is a projection problem from geometry, where the conditions afforded by contract lenses are essential for its construction. The second example concerns bidirectional data conversion, specifically, string processing and formatting. It showcases that within our framework, such computation tasks can be constructed in a point-free style, of which efficiency are guaranteed by calculational laws. The third example stems from a classic scenario of program calculation, it demonstrates the ability to reason about and optimize complicated bidirectional programs through semantics-preserving transformation based on calculational laws, in a way that one would have done for unidirectional programs.

### 7.1 Projection onto a Hyperplane

Let us look at an example to see the expressive power of contract lenses, especially how we can use contracts to constrain the changes of source and view. One basic computation in the area of geometry is to calculate the projection of a point onto a hyperplane in a higher dimensional Euclidean space. In this example, we want to synchronize a point  $xs = [x_1, x_2, \dots, x_n]$ <sup>11</sup> in a  $n$ -dimensional Euclidean space with the projection of it onto the hyperplane  $H : \sum_{i=1}^n x_i = 0$ . The projection of  $X$  onto  $H$  is the point  $ys = [x_1 - m, x_2 - m, \dots, x_n - m]$  where  $m = \frac{1}{n} \sum_{i=1}^n x_i$ . What's more, there is a unique hyperplane  $H'$  parallel to  $H$  and through the point  $xs$ . We want an extra property that the new point obtained from backward transformation is on the hyperplane  $H'$ . In other words, the task is to synchronize a list of numbers with the differences between each number and the mean of all numbers, meanwhile the mean of the source list is unchanged after changes on the view list.

One way to implement this synchronization using lenses is to compose two lenses, where one lens synchronizes a list with a pair of the list itself and its mean, and the other lens synchronizes this pair with the list of differences. The constraints that the dimension  $n$  and the hyperplane  $H'$  should not be changed can be easily expressed with contracts. The full implementation is as follows:

```

1076   bproj : [Float] ↔ [Float]
1077   bproj = bmean; bdiff
1078
1079   bmean : [Float] ↔ (Float, [Float])
1080   bmean = CLens g p cs' cv'
1081     where g xs                = (mean xs, xs)
1082           p (m, xs')        = xs'
1083           cs' xs xs'        = mean xs = mean xs' ∧ eqlength xs xs'
1084           cv' (m, xs) (m', xs') = m = m' = mean xs = mean xs' ∧ eqlength xs xs'
1085
1086   bdiff : (Float, [Float]) ↔ [Float]
1087   bdiff = CLens g p cs' cv'
1088     where g (m, xs)          = map (+(-m)) xs
1089           p (m, -) xs'       = (m, map (+m) xs')
1090           cs' (m, xs) (m', xs') = m = m' = mean xs = mean xs' ∧ eqlength xs xs'
1091           cv' xs xs'         = sum xs' = 0 ∧ eqlength xs xs'
1092
1093   mean : [Float] → Float
1094   mean = λxs → sum xs / fromIntegral (length xs)

```

The specifications of synchronization behaviour on each lenses are clearly expressed by contracts, which enables the compositions as we see in the definition of *bproj*.

### 7.2 String Formatting and Processing

Specifying programs that manipulate texts/strings bidirectionally is not new, and has been extensively studied in Bohannon et al. (2008); Matsuda and Wang (2015). The novelty

<sup>11</sup> Here we use a list of length  $n$  to represent a point in  $n$ -dimensional space.



of our framework is that it supports a point-free style of specifications and calculational reasonings for such computational tasks.

### 7.2.1 String Formatting

Let us look at the following string formatting task: given an input string, we want to filter out all digits, and convert all remaining characters to upper case. With contract lens combinators, we readily specify it in point-free style (for simplicity, we assume that characters in strings are either numbers or letters):

```

1105
1106
1107
1108
1109
1110
1111
1112
1113   bformatting : String ↔ String
1114   bformatting = bfilter (not ∘ isDigit); bmap' btoUpper
1115   where
1116     btoUpper :: Char ↔ Char
1117     btoUpper = CLens toUpper putToLower cs cv
1118     cs = λ _ c → not (isDigit c)
1119     cv = λ _ c → isUpper c
1120     putToLower x y = if isUpper x then y else toLower y

```

The composition is valid, since one can check that  $fcond (not \circ isDigit)$  and  $licond (\lambda _ c \rightarrow not (isDigit c))$  are by definition equivalent.

In this naive specification, intermediate structures are created after one lens, and are immediately consumed by another, in both directions. Recall that  $bfilter$  is an instance of  $bfoldr'$ , using **BFOLDER' FUSION**, we reason as follows:

```

1127
1128   bformatting
1129 = { definition }
1130   bfilter (not ∘ isDigit); bmap' btoUpper
1131 = { expressing bfilter as bfoldr' }
1132   bfoldr' ctrue (fcond (not ∘ isDigit)) (bfilterAlg (not ∘ isDigit)); bmap' btoUpper
1133 = { BFOLDER' FUSION }
1134   bfoldr' ctrue (licond (λ _ c → isUpper c)) balg

```

where

```

1135
1136   balg :: (Either () (Char, [Char])) ↔ [Char]
1137   balg = CLens g p cs cv
1138   where g (Left ()) = []
1139         g (Right (x, xs)) = if not (isDigit x) then toUpper x : xs else xs
1140   p (Left ()) _ = Left ()
1141   p (Right (x, xs)) (x' : xs') = if not (isDigit x)
1142     then Right (putToLower x x', xs')
1143     else Right (x, x' : xs')
1144   p (Right (x, [])) [] = if not (isDigit x) then Left () else Right (x, [])
1145   cs = lift ctrue (licond (λ _ c → isUpper c))
1146   cv = licond (λ _ c → isUpper c)
1147
1148
1149
1150

```

The definition of *balg* is not as complicated as it seems: it is essentially the combination of *bfilterAlg* (*not*  $\circ$  *isDigit*) and *btoUpper*.

It is easy to verify the condition of the **BFOLDR' FUSION** law, which is the lens equivalence relation

$$\begin{aligned} & \text{bfilterAlg } (\text{not} \circ \text{isDigit}); \text{bmap}' \text{ btoUpper} = \text{blistF}' \text{ true} \\ & (\text{bmap}' \text{ btoUpper}); \text{balg} \end{aligned}$$

The calculated version creates no intermediate structure and hence is more efficient in practice.

### 7.2.2 String Encoding and Decoding

Another useful string processing algorithm is the encoding and decoding, which is usually used in compressing a string. It is very appropriate to write them as a single bidirectional program in order to make it easier to maintain and optimize the encoding and decoding algorithms at the same time (Matsuda and Wang, 2020). Let us consider the following simple string encoding algorithm which illustrates the idea of Run Length Encoding.

$$\begin{aligned} & \text{compression} : [\text{String}] \rightarrow [\text{Int}] \\ & \text{compression} = \text{foldr}' \text{ cat} \circ \text{map } \text{ascii} \circ \text{map } \text{encode} \\ & \quad \text{where } \text{encode} \quad \quad \quad = (\text{head } \text{ws}, \text{length } \text{ws}) \\ & \quad \quad \text{ascii } (x, y) \quad \quad = (\text{ord } x, y) \\ & \quad \quad \text{cat } (\text{Left } ()) \quad \quad = [] \\ & \quad \quad \text{cat } (\text{Right } ((x, y), b)) = x : y : b \end{aligned}$$

For simplicity, the input string has already been splitted into a list of strings, where each string consists of consecutive identical characters. The *compression* compresses consecutive identical characters into its ASCII value and number of consecutive occurrences. The *map encode* maps the consecutive identical characters to the pair of the character and the length. Then the *map ascii* transforms the characters to their ASCII values. Finally, the *foldr' cat* concatenates the pairs to a single list. For example, <sup>12</sup>

$$\text{compression } ["aaaaa", "bbbb", "cccccccc"] = [97, 5, 98, 4, 99, 9]$$

Using the contract-lens combinators we defined in [Section 5](#), it is easy to derive a bidirectional version of the function *compression*. The length of the results should not be changed, meanwhile the ASCII values in the results should all be greater than or equal to 0 and less than 128. Thus, the view condition is defined as  $\text{cv}_{\text{comp}} \text{vas} = (|v| = |as|) \wedge (\forall 1 \leq i \leq |as|, \text{odd } i \vee (0 \leq as_i < 128))$ .

$$\begin{aligned} & \text{bcompression} : [\text{String}] \leftrightarrow [\text{Int}] \\ & \text{bcompression} = \text{bmap}' \text{ bencode} \\ & \quad \quad \quad ; \text{bmap}' \text{ bascii} \\ & \quad \quad \quad ; \text{bfoldr}' (\lambda\_ (x, -) \rightarrow 0 \leq x < 128) \text{cv}_{\text{comp}} \text{bcat} \end{aligned}$$

where the following contract lenses are used

<sup>12</sup> The ASCII value of 'a' is 97, 'b' is 98, 'c' is 99. We assume that the *Char* type only includes the standard 128 ASCII values for simplicity.

```

1197 bencode : String  $\leftrightarrow$  (Char, Int)
1198 bencode = CLens ( $\lambda ws \rightarrow (\text{head } ws, \text{length } ws)$ ) ( $\lambda _ (a, n) \rightarrow \text{replicate } n a$ )
1199           ( $\lambda _ as \rightarrow \text{allsame } as$ ) ctrue
1200   where allsame xs = (xs = "")  $\vee$  (and $ map (= head xs) (tail xs))
1201 bascii : (Char, b)  $\leftrightarrow$  (Int, b)
1202 bascii = CLens ( $\lambda (x, y) \rightarrow (\text{ord } x, y)$ ) ( $\lambda _ (x, y) \rightarrow (\text{chr } x, y)$ )
1203           ctrue ( $\lambda _ (x, _) \rightarrow 0 \leq x < 128$ )
1204 bcat : Either () ((Int, Int), [Int])  $\leftrightarrow$  [Int]
1205 bcat = CLens g p cs cv
1206   where g (Left ()) = []
1207           g (Right ((x, y), b)) = x : y : b
1208           p (Left ()) [] = Left ()
1209           p (Right _) (x : y : b) = Right ((x, y), b)
1210           cs = lift ( $\lambda _ (x, _) \rightarrow 0 \leq x < 128$ ) cvcomp
1211           cv = cvcomp

```

It is easy to check the contract lens *bcompression* is well-defined. However, this version of *bcompress* is not so efficient because it traverses the string three times. We can use the bidirectional calculation laws in Section 6 to reduce both the compression and the decompression algorithms to only one traversal simultaneously.

```

1217   bcompression
1218 = { definition }
1219   bmap' bencode
1220   ; bmap' bascii
1221   ; bfoldr' ( $\lambda _ (x, _) \rightarrow 0 \leq x < 128$ ) cvcomp bcat
1222 = { BMAP' FUSION }
1223   bmap' (bencode; bascii)
1224   ; bfoldr' ( $\lambda _ (x, _) \rightarrow 0 \leq x < 128$ ) cvcomp bcat
1225 = { BFOLDER' -BMAP FUSION }
1226   bfoldr' ( $\lambda _ as \rightarrow \text{allsame } as$ ) cvcomp (bmapF (bencode; bascii); bcat)

```

### 7.3 Bidirectional Maximum Segment Sum

Now let us turn to another example involving more advanced program calculation. The maximum segment sum is a classic problem in the area of program calculation. To demonstrate the ability of our calculation framework, we change the specification of *mss* in Section 2.3 into a bidirectional version directly using contract-lens combinators, and optimize it to a more efficient version which has time complexity  $O(n)$  in both get and put directions, meanwhile the semantics is preserved.

To see this concretely, let us first get a bidirectional version of *mss* without considering efficiency. To achieve this, we introduce a refinement type *TailsList*  $a = \{as : [[a]] \mid (\forall 1 \leq i < n, \text{tail } as_i = as_{i+1}) \wedge (\text{tail } as_n = [])\}$ . It is a modified version of the type  $[[a]]$ , where each element of the list is the tail of the previous element, and the tail of the last element is the empty list. The specification of the bidirectional version of *mss* is

```

1240
1241
1242

```

```

1243  $bmss : [Int] \leftrightarrow Int$ 
1244  $bmss = binits$ 
1245   ;  $bmapl \tilde{c}v_1 \tilde{c}v_2 [] btails_{init}$ 
1246   ;  $bmapl \tilde{c}v_2 \tilde{c}v_3 [] bmapSum$ 
1247   ;  $bmapl \tilde{c}v_3 \mathit{ctrue} [] bmaximum2$ 
1248   ;  $bmaximum'$ 

```

where the definitions of the contracts and contract lenses appeared are

```

1250  $\tilde{c}v_1 = \lambda a a' \rightarrow \mathit{init} a' = a$ 
1251  $\tilde{c}v_2 = \lambda b b' \rightarrow \mathit{map} \mathit{init} (\mathit{init} b') = b$ 
1252  $\tilde{c}v_3 = \lambda b b' \rightarrow \mathit{map} (+(-\mathit{last} b')) (\mathit{init} b') = b$ 
1253

```

```

1254  $btails_{init} : [Int] \Rightarrow [Int] \leftrightarrow \mathit{TailsList} Int$ 
1255  $btails_{init} a = \mathit{CLens} \mathit{tails}' (\lambda _v \rightarrow \mathit{head} v) \mathit{cs} \mathit{cv}$ 
1256   where  $\mathit{cs} = \lambda _a' \rightarrow \tilde{c}v_1 a a'$ 
1257          $\mathit{cv} = \lambda _b' \rightarrow \tilde{c}v_2 (\mathit{tails}' a) b'$ 
1258

```

```

1259  $bmapSum : \mathit{TailsList} Int \Rightarrow \mathit{TailsList} Int \leftrightarrow [Int]$ 
1260  $bmapSum a = \mathit{CLens} (\mathit{map} \mathit{sum}) p \mathit{cs} \mathit{cv}$ 
1261   where  $p \_xs = \mathit{map} (\lambda t \rightarrow t \# [\mathit{last} xs]) a \# [[\mathit{last} xs]]$ 
1262          $\mathit{cs} = \lambda _a' \rightarrow \tilde{c}v_2 a a'$ 
1263          $\mathit{cv} = \lambda _b' \rightarrow \tilde{c}v_3 (\mathit{map} \mathit{sum} a) b'$ 
1264

```

```

1265  $bmaximum2 : [Int] \Rightarrow [Int] \leftrightarrow Int$ 
1266  $bmaximum2 a = \mathit{CLens} \mathit{maximum} p \mathit{cs} \mathit{ctrue}$ 
1267   where  $p \_x = \mathbf{let} t = a \# [0] \mathbf{in} \mathit{map} (+ (x - \mathit{maximum} t)) t$ 
1268          $\mathit{cs} = \lambda _a' \rightarrow \tilde{c}v_3 a a'$ 
1269

```

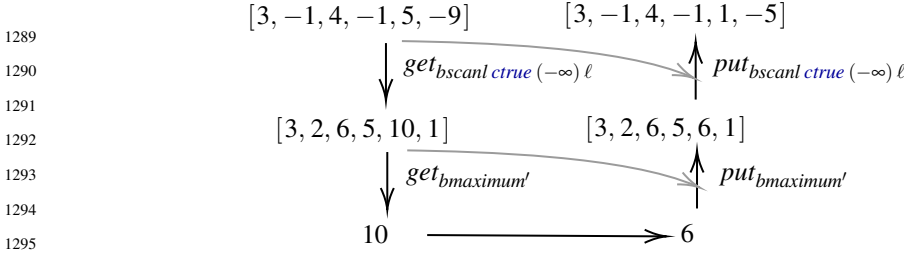
The  $binits$  and  $bmaximum'$  have been already defined in the previous sections. It is easy to check that  $bmss$  is well-defined, i.e., satisfies round-tripping properties and the condition of lens composition.

Next, we make use of the bidirectional calculation rules we developed in [Section 6](#) to optimize the  $bmss$ . The calculation goes as follows.

```

1275  $bmss$ 
1276 = { definition }
1277  $binits$ 
1278   ;  $bmapl \tilde{c}v_1 \tilde{c}v_2 [] btails_{init}$ 
1279   ;  $bmapl \tilde{c}v_2 \tilde{c}v_3 [] bmapSum$ 
1280   ;  $bmapl \tilde{c}v_3 \mathit{ctrue} [] bmaximum2$ 
1281   ;  $bmaximum'$ 
1282 = { BMAPL FUSION }
1283  $binits$ 
1284   ;  $bmapl \tilde{c}v_1 \mathit{ctrue} [] (btails_{init}; ; bmapSum; ; bmaximum2)$ 
1285   ;  $bmaximum'$ 
1286 = { a specific bidirectional Horner's rule (to be discussed below) }
1287
1288

```

Fig. 3. Visualization of an example calculation of  $bms$ .

```

1301   binit
1302   ; bmapl  $\tilde{c}v_1$  ctru [] (bfoldlinit ctru  $(-\infty)$   $\ell$ )
1303   ; bmaximum'
1304   = { BIDIRECTIONAL SCAN LEMMA }
1305   bscanl ctru  $(-\infty)$   $\ell$ 
1306   ; bmaximum'

```

One thing worth noting is that in the third step of calculation we use a specific bidirectional Horner's rule:

```

1310   btailsinit; ; bmapSum; ; bmaximum2 = bfoldlinit ctru  $(-\infty)$   $\ell$ 
1311   where
1312      $\ell : Int \Rightarrow Either Int (Int, Int) \leftrightarrow Int$ 
1313      $\ell b = CLens\ g\ p\ cs\ true$ 
1314     where  $g\ (Left\ ()) = -\infty$ 
1315              $g\ (Right\ (x, y)) = \max\ (x + y)\ x$ 
1316              $p\ _t = Right\ (t - \max\ b\ 0, b)$ 
1317              $cs = \lambda\_t' \rightarrow t' = Right\ (_, b)$ 

```

The get direction of  $(btails_{init}; ; bmapSum; ; bmaximum2)$   $a$  for any  $a : [Int]$  is similar to the original Horner's rule with  $\otimes = +$  and  $\oplus = \max$ . It would take space to develop a general bidirectional Horner's rule for any  $\oplus$  and  $\otimes$ , because we require that  $\oplus$  and  $\otimes$  form a ring structure and keep it in the bidirectional setting. However, it is useful to define and prove some specific bidirectional versions of the Horner's rule like this.

By now, we have successfully derived a correct and linear-time efficient bidirectional program that can synchronize a list with its maximum segment sum.

Let us look at an example to get a better understanding of our final result  $bscanl\ (-\infty)\ \ell; bmaximum'$  that is visualized in Figure 3. Given the input list  $xs = [3, -1, 4, -1, 5, -9]$ ,  $get_{bscanl\ true\ (-\infty)\ \ell}\ xs$  yields  $[3, 2, 6, 5, 10, 1]$ , whose each element refers to the maximum segment sum ending at this position. Then,  $get_{bmaximum'}\ [3, 2, 6, 5, 10, 1]$  yields 10, which is the maximum segment sum of the whole list. Now we change the result from 10 to 6. For the backward direction,  $put_{bmaximum'}\ [3, 2, 6, 5, 10, 1]\ 6$  yields  $[3, 2, 6, 5, 6, 1]$ . Finally,  $put_{bscanl\ true\ (-\infty)\ \ell}\ [3, -1, 4, -1, 5, -9]\ [3, 2, 6, 5, 6, 1]$  yields  $[3, -1, 4, -1, 1, -5]$ .

## 8 Related Work

In this section, we discuss related work on partiality in the lens framework, Hoare-style reasoning of BX, automatic bidirectionalization, and some attempts on calculating with lenses.

### 8.1 Lens Family and Partiality of *put*

The most prominent approach to bidirectional transformation is the lens framework formally introduced by Foster et al. (2007). It is highly influential and directly inspired a number of follow-on works including Boomerang (Bohannon et al., 2008), quotient lenses (Foster et al., 2008), matching lenses (Barbosa et al., 2010), symmetric lenses (Hofmann et al., 2011), edit lenses (Hofmann et al., 2012), BiGUL (Ko et al., 2016), applicative lenses (Matsuda and Wang, 2015), HOBiT (Matsuda and Wang, 2018) and so on. The present paper on contract lenses is no exception. On the issue of partiality, different approaches were taken by the various works, which can be broadly categorized into the following.

#### 8.1.1 Formulation of Contracts and Relation to Type Systems

As argued in Section 1.1, giving total definitions to *get* and *put* components is not always desirable, as the effort in achieving it necessarily complicates program design and reasoning. Some previous work on lenses ensures the totality of them by advanced type systems, with enriched type constraints over the type variables  $S$ ,  $V$  in the lens type  $S \leftrightarrow V$ . For example, in Foster et al. (2007), partial lenses are ruled out by set-based type constraints that precisely characterize the domain/range of *get* and *put*, and in Boomerang (Bohannon et al., 2008), the underlying String type is enriched with regular languages to serve as types for dictionary lenses.

As far as we know, lens formulations with enriched type systems like the above are not readily used to flexibly express the bidirectional behaviours we see in this paper. Take  $bmap : (S \leftrightarrow V) \rightarrow [S] \leftrightarrow [V]$  as an example. With contracts, we can easily ensure that the changes on view do not modify the length of lists by setting the view condition to *eqlength*. However, it is non-trivial to express the “equal length” view condition by only constraining the types  $S$  and  $V$  themselves, instead of specifying constraints on the changes of values of types  $S$  and  $V$ . By adding an additional parameter to *bmap* specifying the length of the source and view list, one could encode *bmap* indirectly with a notion of dependent/refinement types into something like the following.

$$bmap : (n : \mathbb{N}) \rightarrow (S \leftrightarrow V) \rightarrow (\{xs : [S] \mid |xs| = n\} \leftrightarrow \{ys : [V] \mid |ys| = n\})$$

This version of *bmap* fixes the length of lists, which is obviously less general than the versions using *eqlength* like the *bmap* in Section 5.2.1 and *bmap'* in Section 5.2.2.

The “equal length” view condition is essentially a constraint on the dynamic changes of inputs to a lens, which can be nicely handled by our view contract. In our framework, contracts specifies the ranges that lens components behave well, the dynamic changes that a lens can reasonably accept, and the conditions that different components can compose together.

1381 It is worth noting that different from the previous work on constraining the source and  
 1382 view types (Foster et al., 2007; Bohannon et al., 2008), contracts are not part of types, but  
 1383 rather additional specifications that parallel *get* and *put*. Moreover, users have full control  
 1384 of these specifications, just as how they specify the component *get* and *put* in the first  
 1385 place. In this sense, user have the flexibility to choose different contracts based on the  
 1386 same underlying *get* and *put*. For instance, the “equal length” condition for *bmap* may be  
 1387 strengthened so that additionally the first element of the list is preserved. These choices are  
 1388 completely up to the users.

1389 An alternative design choice of contract lenses is to encode the **BACKWARDVALIDITY**  
 1390 and **FORWARDVALIDITY** laws as well as the extra conditions of the  
 1391 **CONDITIONEDPUTGET** and **CONDITIONEDGETPUT** laws directly into the types of  
 1392 *get* and *put* with refinement types.

$$1393 \quad \begin{aligned} & \textit{get} : \{s : S \mid cs\ s\ s\} \rightarrow \{v : V \mid cv\ v\ v\} \\ & \textit{put} : (s : S) \rightarrow \{v : V \mid cv\ (\textit{get}\ s)\ v\} \rightarrow \{s' : S \mid cs\ s\ s'\} \end{aligned}$$

1395 With the above refinement type signatures, we can use the original **PUTGET** and **GETPUT**  
 1396 laws of lenses. Note that the definition of contract lenses is still a four-tuple of *get*,  
 1397 *put*, *cs* and *cv* in this case. There is no clear advantage or disadvantage between these  
 1398 two approaches. We choose to characterize the properties of contracts with explicit laws  
 1399 like **BACKWARDVALIDITY** and **FORWARDVALIDITY** to avoid the complication of type  
 1400 signatures and emphasize the differences between traditional lenses and contract lenses.

1401 In this work, we do not impose any restriction on the constraints used in contracts. It  
 1402 is the users’ work to prove the round-tripping properties of contract lenses and the well-  
 1403 definedness of lens composition by either handwritten proofs or formalisation in theorem  
 1404 provers like Agda. As a result, the designer of a practical system that implements contract  
 1405 lenses has to strike a balance between expressiveness of contracts and checkability of con-  
 1406 tracts implications. Nonetheless, we believe such systems are implementable, by restricting  
 1407 the set of contracts available to users to a small set of efficiently solvable constraints. As  
 1408 shown in our examples, simple predicates like *eqlength* can already help with constructing  
 1409 powerful combinators like generic mapping over lists.

### 1411 8.1.2 Edit Lenses

1412 Edit lenses (Hofmann et al., 2012) model changes to view/source as operations (edits)  
 1413 in contrast to states in the traditional lenses. The edits are represented as monoids, and  
 1414 monoid actions on set become the actions of applying an edit to a state. As a result, only  
 1415 the edits in the monoid are allowed to be applied to the states, which in a way restricts  
 1416 changes to the source and view. But unlike contract lenses, these restrictions are not used  
 1417 to address partiality; in fact edit lenses have the same problem of partiality as state-based  
 1418 ones because the monoid actions are allowed to be partial. For example, the edit *del* which  
 1419 deletes the last element of a list is partial as we can not apply it to an empty list. Extra  
 1420 dynamic checks are needed to ensure that the computation of edit lenses will not fail. For  
 1421 contract lenses, the *get* and *put* will not fail as long as the source conditions and view  
 1422 conditions are satisfied.

1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426

### 8.1.3 Totality with Maybe Monad

Another approach is to wrap the return type of *get* and *put* in the *Maybe* monad to remove partiality (Matsuda and Wang, 2015; Ko et al., 2016; Xia et al., 2019). The *put* direction is a total function of type  $s \rightarrow v \rightarrow \text{Maybe } s$  and it returns *Nothing* at run-time when an invalid input is passed to it. This approach is unsuitable for program calculation as it lacks the ability to reason about partiality statically. We want to know the static specification of a program and get meaning results instead of just getting a *Nothing* when the program fails. Moreover, the specification can guide the design of program calculation laws.

### 8.1.4 Other Discussions

The properties of partial BX and the relations between them are discussed extensively in Stevens (2014). Different from our goal, the discussion there does not concern practical program construction nor mentions composition of transformations. In contrast, we focus on lenses that satisfy the round-tripping property on possibly partial domains. We make partiality explicit as a component of lenses, and use it to explore composition behaviour of partial lenses.

## 8.2 Hoare-style Reasoning of Bidirectional Transformation

In Ko and Hu (2018), a reasoning framework for BiGUL programs based on Hoare logic is proposed, which is able to precisely characterize the bidirectional behaviours by reasoning in the *put* direction. The main concept is the *put* triplet in the form of  $\{R\}b\{R'\}$ , which includes a set of pre- and post-conditions that are used to reason about the behaviour of *put* in a way similar to the Hoare logic: if the original source  $s$  and the updated view  $v$  satisfy the precondition  $R$ , then  $\text{put}_b s v$  will produce an updated source satisfying the postcondition  $R'$ .

To some extent, their pre- and post-conditions serve a similar purpose to our **BACKWARDVALIDITY** law: if the original source  $s$  and the updated view  $v$  satisfy the view condition  $cv (get s) v$ , then  $\text{put } s v$  will successfully produce an updated source satisfying the source condition  $cs s (put s v)$ . However, the novelty of contract lenses does not solely rely on the **BACKWARDVALIDITY** law, but also the combination with other three laws of the round-tripping properties which give a clear specification of lenses to resolve the partiality problem and make the composition of contract lenses easy and well-behaved. It is worth mentioning that in their framework reasoning about lens composition is difficult and involves several complicated proof rules. In contrast, contract lenses make such reasoning easy: two lenses  $\ell_1 : \{cs_{\ell_1}\} S \leftrightarrow V \{cv_{\ell_1}\}$  and  $\ell_2 : \{cs_{\ell_2}\} V \leftrightarrow T \{cv_{\ell_2}\}$  can be composed into a lens  $\ell_1; \ell_2 : \{cs_{\ell_1}\} S \leftrightarrow T \{cv_{\ell_2}\}$  given the condition proposed in Definition 2.

Furthermore, the purpose of pre- and post-conditions differs from that of source and view conditions. While pre- and post-conditions mainly focus on specifying the behaviours of the *put* components, our primary objective is to address the partiality problem of lenses, which allows for straightforward design of lenses and calculation laws.



### 8.3 Bidirectionalization

1473 Bidirectionalization is an approach to bidirectional programming that is different from the  
 1474 lens framework. Instead of writing bidirectional programs directly in a specialized lan-  
 1475 guage, it aims to mechanically convert existing unidirectional programs into bidirectional  
 1476 ones. Voigtländer (2009) gives a high-order function *bff* that receives a polymorphic *get*  
 1477 function, and returns its *put* counterpart. The technique is extended (Voigtländer et al.,  
 1478 2010) by combining it with syntactic bidirectionalization (Matsuda et al., 2007), which  
 1479 separates view changes in shape and in content. However, bidirectionalization is done  
 1480 for whole programs which lacks modular reasoning of compositions, and therefore is not  
 1481 suitable for program calculation.  
 1482

### 8.4 Calculating with Lenses

1483  
 1484  
 1485  
 1486 The goal of *generic point-free* lenses (Pacheco and Cunha, 2010) is the most similar to  
 1487 ours. In that work, lens combinators are designed for many traditional high-order functions  
 1488 including *fold* and *map*. Subsequently, the point-free lenses are use for a limited form  
 1489 of calculation where the universal property (uniqueness) of the lens version of *fold* was  
 1490 proved and used to establish some program calculation laws for lenses such as the *fold-map*  
 1491 fusion (Pacheco and Cunha, 2011).

1492 But very different from ours, their work is based on the traditional lenses without con-  
 1493 tracts, which means that the problem of partiality seriously limits the composition of lenses.  
 1494 As a result, many crucial calculation laws such as the **SCAN LEMMA** are not expressible  
 1495 in their framework.  
 1496

## 9 Formalisation with Agda

1497  
 1498  
 1499  
 1500 In this section, we briefly discuss one possible formalisation of contract lenses in Agda.  
 1501 We use this formalisation to prove the correctness of lens composition, all lens combina-  
 1502 tors, all calculation laws and most of the examples (except the string processing example  
 1503 in Section 7.2) in this paper. As mentioned in Section 1.2, our intention is not to restrict  
 1504 potential users of contract lenses within this formalisation, but rather to provide a calcu-  
 1505 lation framework which allows any method of reasoning. This Agda formalisation shows  
 1506 one potential way to mechanise our framework.

1507 The formalisation of the whole contract lens calculation framework is rather straight-  
 1508 forward. A contract lens is a (possibly mutually defined) four-tuple *get*, *put*, *cs* and *cv*,  
 1509 with a set of laws on them. This construction is formalised faithfully in the Agda code,  
 1510 where we define the lens type as a record type

```

1511 record Lens (S : Set) (V : Set) where
1512   field
1513     -- four-tuple
1514     get : S → V
1515     put : S → V → S
1516     cs  : S → S → Set
  
```

1517  
 1518

$cv : V \rightarrow V \rightarrow Set$

-- laws

*BackwardValidity* :  $\forall (a : S) (b : V) \rightarrow cv (get a) b \rightarrow cs a (put a b)$

*ForwardValidity* :  $\forall (a : S) \rightarrow cs a a \rightarrow cv (get a) (get a)$

*PutGet* :  $\forall (a : S) (b : V) \rightarrow cv (get a) b \rightarrow get (put a b) = b$

*GetPut* :  $\forall (a : S) \rightarrow cs a a \rightarrow cv put a (get a) = a$

Typically, to construct an instance of this type, one will first define the four-tuple, and then gives proof of the four laws.

The formalisation of lens combinators also follows from what we have in the paper. For instance, the *bmap* combinator with type

$$bmap : (\{ctrue\} S \leftrightarrow V \{ctrue\}) \rightarrow ([S] \leftrightarrow [V])$$

is formalised in Agda using existential types as

$$bmap : \forall \{S V : Set\} \rightarrow (\exists (S \leftrightarrow V) \lambda \ell \rightarrow cv \ell \Leftrightarrow ctrue \wedge cs \ell \Leftrightarrow ctrue) \rightarrow ([S] \leftrightarrow [V])$$

One difference between our Agda formalisation and what we have in the paper is that the Agda formalisation does not use the syntactic sugar  $\ell : S \Rightarrow (S \leftrightarrow V)$  defined in [Section 5.2.3](#) to restrict the parameterised lens  $\ell$  to have a fixed *get* component. Instead, it defines  $\ell$  as a lens of type  $(S, S) \leftrightarrow (V, V)$ , where the parameter is embedded into the first component of the source pair. The former form is more clear and suitable for human reading, while the latter form is easier to formalise. We provided a translation between these two kinds of lenses and proved its correctness in the Agda formalisation.

For the calculation part of this framework, we defined an equivalence relation between lenses of the above type as described in [Definition 3](#). We also prove the congruence theorem for high-order lenses. Take *bmap* for example, we prove that if  $\ell_1 \sim \ell_2$ , then  $bmap \ell_1 \sim bmap \ell_2$ . Our calculation laws are defined as theorems stating equivalences of lenses.

## 10 Conclusion

In this work, we propose a framework based on program calculation to enable the development of complex but efficient BX programs that are correct by construction. As part of the framework, we design a novel extension to lenses, *contract lenses*, for handling partiality and use it to justify general composition of lenses. Based on this, we extend the theories for program calculation to BX programming by designing combinators to capture bidirectional recursive computation patterns and proving their properties. We look at the list datatype and give proofs for fundamental calculation laws including various fusion laws for bidirectional *fold* and *map* and the bidirectional *scan* lemma. We showcase the construction of a realistic projection program, the derivation of efficient bidirectional string processing programs, and the *maximum segment sum* program to demonstrate the effectiveness of our framework.

This work focuses on the calculation for bidirectional transformations on lists, which mirrors the classic work on the *theory of list* (Bird, 1989, 1987) in the literature of program calculation. Generalizing this bidirectional program calculation framework to algebraic datatypes generated by polynomial functors is a natural next step. Another possible future

work is to design practical systems based on contract lenses to reason about and optimize  
BXs, automating the verification of round-tripping properties and lens composition using  
SMT solvers.

### Conflict of Interest

None.

### References

- Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J. & Stevens, P. (2018) Introduction to bidirectional transformations. In *Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*. Springer International Publishing. Cham. chapter 1, pp. 1–28. Available at: [https://doi.org/10.1007/978-3-319-79108-1\\_1](https://doi.org/10.1007/978-3-319-79108-1_1).
- Bancilhon, F. & Spyrtatos, N. (1981) Update semantics of relational views. *ACM Trans. Database Syst.* **6**(4), 557–575.
- Barbosa, D. M., Cretin, J., Foster, N., Greenberg, M. & Pierce, B. C. (2010) Matching lenses: Alignment and view update. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 193–204.
- Bird, R. S. (1987) An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design*. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 5–42.
- Bird, R. S. (1989) Algebraic identities for program calculation. *Comput. J.* **32**(2), 122–126.
- Bird, R. S. (1989) Lectures on constructive functional programming. *Constructive Methods in Computing Science*. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 151–217.
- Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A. & Schmitt, A. (2008) Boomerang: Resourceful lenses for string data. Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 407–419.
- Bohannon, A., Pierce, B. C. & Vaughan, J. A. (2006) Relational lenses: A language for updatable views. Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. New York, NY, USA. Association for Computing Machinery. p. 338–347.
- Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* **29**(3), 17–es.
- Foster, J. N., Pilkiewicz, A. & Pierce, B. C. (2008) Quotient lenses. Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 383–396.
- Gibbons, J. (2002) Calculating functional programs. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer-Verlag. pp. 148–203.
- Gibbons, J. (2011) Maximum segment sum, monadically (distilled tutorial). *Electronic Proceedings in Theoretical Computer Science*. **66**, 181–194.
- Gill, A., Launchbury, J. & Peyton Jones, S. L. (1993) A short cut to deforestation. Proceedings of the Conference on Functional Programming Languages and Computer Architecture. New York, NY, USA. Association for Computing Machinery. p. 223–232.
- He, X. & Hu, Z. (2018) Putback-based bidirectional model transformations. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA. Association for Computing Machinery. p. 434–444.

- 1611 Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K. & Nakano, K. (2010) Bidirectionalizing graph  
1612 transformations. Proceedings of the 15th ACM SIGPLAN International Conference on Functional  
1613 Programming. New York, NY, USA. Association for Computing Machinery. p. 205–216.
- 1614 Hofmann, M., Pierce, B. & Wagner, D. (2011) Symmetric lenses. Proceedings of the 38th Annual  
1615 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY,  
1616 USA. Association for Computing Machinery. p. 371–384.
- 1617 Hofmann, M., Pierce, B. & Wagner, D. (2012) Edit lenses. Proceedings of the 39th Annual ACM  
1618 SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA.  
1619 Association for Computing Machinery. p. 495–508.
- 1620 Hu, Z., Iwasaki, H. & Takeichi, M. (1996) Deriving structural hylomorphisms from recursive  
1621 definitions. Proceedings of the First ACM SIGPLAN International Conference on Functional  
1622 Programming. New York, NY, USA. Association for Computing Machinery. p. 73–82.
- 1623 Ko, H.-S. & Hu, Z. (2018) An axiomatic basis for bidirectional programming. Proceedings of the  
1624 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.  
1625 New York, NY, USA. Association for Computing Machinery.
- 1626 Ko, H.-S., Zan, T. & Hu, Z. (2016) Bigul: A formally verified core language for putback-based bidi-  
1627 rectional programming. Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation  
1628 and Program Manipulation. New York, NY, USA. Association for Computing Machinery. p.  
1629 61–72.
- 1630 Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2007) Bidirectionalization trans-  
1631 formation based on automatic derivation of view complement functions. Proceedings of the 12th  
1632 ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA.  
1633 Association for Computing Machinery. p. 47–58.
- 1634 Matsuda, K. & Wang, M. (2015) Applicative bidirectional programming with lenses. Proceedings of  
1635 the 20th ACM SIGPLAN International Conference on Functional Programming. New York, NY,  
1636 USA. Association for Computing Machinery. p. 62–74.
- 1637 Matsuda, K. & Wang, M. (2015) "bidirectionalization for free" for monomorphic transformations.  
1638 *Sci. Comput. Program.* **111**(P1), 79–109.
- 1639 Matsuda, K. & Wang, M. (2018) Hobit: Programming lenses without using lens combinators.  
1640 European Symposium on Programming. Springer. pp. 31–59.
- 1641 Matsuda, K. & Wang, M. (2020) Sparcl: a language for partially-invertible computation. *Proc. ACM*  
1642 *Program. Lang.* **4**(ICFP), 118:1–118:31.
- 1643 Pacheco, H. & Cunha, A. (2010) Generic point-free lenses. Proceedings of the 10th International  
1644 Conference on Mathematics of Program Construction. Berlin, Heidelberg. Springer-Verlag. p.  
1645 331–352.
- 1646 Pacheco, H. & Cunha, A. (2011) Calculating with lenses: Optimising bidirectional transforma-  
1647 tions. Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program  
1648 Manipulation. New York, NY, USA. Association for Computing Machinery. p. 91–100.
- 1649 Stevens, P. (2008) A landscape of bidirectional model transformations. Generative and  
1650 Transformational Techniques in Software Engineering II: International Summer School, GTTSE  
1651 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. Berlin, Heidelberg. Springer Berlin  
1652 Heidelberg. pp. 408–424.
- 1653 Stevens, P. (2014) Bidirectionally tolerating inconsistency: Partial transformations. Fundamental  
1654 Approaches to Software Engineering. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 32–46.
- 1655 Stevens, P. (2020) Maintaining consistency in networks of models: bidirectional transformations in  
1656 the large. *Software and Systems Modeling.* **19**(1), 39–65.
- 1657 Tran, V.-D., Kato, H. & Hu, Z. (2020) Birds: Programming view update strategies in datalog. 46th  
1658 International Conference on Very Large Data Bases. VLDB Endowment. p. 2897–2900.
- 1659 Tsigkanos, C., Li, N., Jin, Z., Hu, Z. & Ghezzi, C. (2020) Scalable multiple-view analysis of  
1660 reactive systems via bidirectional model transformations. Proceedings of the 35th IEEE/ACM  
1661 International Conference on Automated Software Engineering. New York, NY, USA. Association  
1662 for Computing Machinery. p. 993–1003.
- 1663 Voigtländer, J. (2009) Bidirectionalization for free! (pearl). Proceedings of the 36th Annual ACM

SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 165–176.

Voigtländer, J., Hu, Z., Matsuda, K. & Wang, M. (2010) Combining syntactic and semantic bidirectionalization. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 181–192.

Xia, L.-y., Orchard, D. & Wang, M. (2019) Composing bidirectional programs monadically. European Symposium on Programming. Springer. pp. 147–175.

1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702

## 1 Calculating with Total Lenses

It is possible to make *bmap* total:

$$\begin{aligned}
 & \mathit{bmap}_{total} : A \rightarrow (A \leftrightarrow B) \rightarrow ([A] \leftrightarrow [B]) \\
 & \mathit{bmap}_{total} a_0 \ell = \mathit{Lens} (\mathit{map} \mathit{get}_\ell) p \\
 & \textbf{where } p \_ \quad [] = [] \\
 & \quad p (x : xs) (y : ys) = \mathit{put}_\ell x y : p xs ys \\
 & \quad p [] \quad (y : ys) = \mathit{put}_\ell a_0 y : p [] ys
 \end{aligned}$$

The additional parameter  $a_0$  is used as a default source value.

One can develop an associated map fusion law for it:

$$\frac{\mathit{get}_{\ell_1} a_0 = b_0}{\mathit{bmap}_{total} a_0 \ell_1; \mathit{bmap}_{total} b_0 \ell_2 = \mathit{bmap}_{total} a_0 (\ell_1; \ell_2)} \quad \text{BMAPTOTAL FUSION}$$

However, this law requires  $\mathit{get}_{\ell_1} a_0 = b_0$ , a semantic condition on default values, which is an unwanted proof obligation to program calculators and optimisers.

## 2 Equivalent Implementation of Combinators

This appendix shows the code for equivalent implementations of some contract-lens combinators in [Section 5](#).

### 2.1 Efficient *bfoldr*

This section shows an efficient implementation of *bfoldr*.

$$\begin{aligned}
 & \mathit{bfoldr} \ell = \mathit{CLens} (\mathit{foldr}' \mathit{get}_\ell) p \mathit{ctrue} \mathit{ctrue} \\
 & \textbf{where } p \textit{ as } b' = \mathit{let} \textit{ bs} = \mathit{tail} (\mathit{scanr} (\lambda a b \rightarrow \mathit{get}_\ell (\mathit{Right} (a, b))) (\mathit{get}_\ell (\mathit{Left} ())) \textit{ as}) \\
 & \quad \textbf{in } \mathit{go} \textit{ as } \textit{ bs } b' \\
 & \quad \mathit{go} [] [] b' = \mathit{case} \mathit{put}_\ell (\mathit{Left} ()) b' \textbf{ of} \\
 & \quad \quad \mathit{Left} () \rightarrow [] \\
 & \quad \quad \mathit{Right} (a', \textit{ bim}') \rightarrow a' : \mathit{go} [] [] \textit{ bim}' \\
 & \quad \mathit{go} (a : \textit{ as}) (\textit{ bim} : \textit{ bs}) b' = \mathit{case} \mathit{put}_\ell (\mathit{Right} (a, \textit{ bim})) b' \textbf{ of} \\
 & \quad \quad \mathit{Left} () \rightarrow [] \\
 & \quad \quad \mathit{Right} (a', \textit{ bim}') \rightarrow a' : \mathit{go} \textit{ as } \textit{ bs } \textit{ bim}'
 \end{aligned}$$

### 2.2 Implementation of *bmap* and *bmap'* with *bfoldr'*

This section shows how to use *bfoldr'* to implement *bmap* and *bmap'*.

$$\begin{aligned}
 & \mathit{bmap} : (\{\mathit{ctrue}\} S \leftrightarrow V \{\mathit{ctrue}\}) \rightarrow ([S] \leftrightarrow [V]) \\
 & \mathit{bmap} \ell = \mathit{bfoldr}' \mathit{ctrue} \mathit{eqlength} \ell' \\
 & \textbf{where} \\
 & \quad \ell' :: \mathit{Either} () (S, [V]) \leftrightarrow [V] \\
 & \quad \ell' = \mathit{CLens} g p (\mathit{lift} \mathit{ctrue} \mathit{eqlength}) \mathit{eqlength} \\
 & \quad g (\mathit{Left} ()) = []
 \end{aligned}$$

$g (\text{Right } (a, bs)) = \text{get}_\ell a : bs$   
 $p (\text{Left } ()) [] = \text{Left } ()$   
 $p (\text{Right } (a, -)) (a' : bs') = \text{Right } (\text{put}_\ell a a', bs')$

$bmap' : (S \leftrightarrow V) \rightarrow ([S] \leftrightarrow [V])$   
 $bmap' \ell = bfoldr' cs_\ell (\text{licond } cv_\ell) \ell'$

**where**

$\ell' :: \text{Either } () (S, [V]) \leftrightarrow [V]$   
 $\ell' = \text{CLens } g p (\text{lift } cs_\ell (\text{licond } cv_\ell)) (\text{licond } cv_\ell)$   
 $g (\text{Left } ()) = []$   
 $g (\text{Right } (a, bs)) = \text{get}_\ell a : bs$   
 $p (\text{Left } ()) [] = \text{Left } ()$   
 $p (\text{Right } (a, -)) (a' : bs') = \text{Right } (\text{put}_\ell a a', bs')$

### 3 Examples for Combinators

This appendix shows examples for some contract-lens combinators and calculation laws in [Section 5](#) and [Section 6](#).

#### 3.1 Computation Instances of $bmaximum$

This section shows two calculation instances of the  $bmaximum$  example in [Section 5.1](#). Let us assume that  $\text{get}_{bmaximum} [9, 2, 5]$  yields 9, and suppose that the output 9 is changed to 4. Now the following calculation shows how this change is reflected back to the input  $[9, 2, 5]$  and get  $[4, 2, 4]$ .

$\text{put}_{bmaximum} [9, 2, 5] 4$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Right } (9, \text{get}_{bmaximum} [2, 5])) 4 = \text{Right } (4, 4) \}$   
 $4 : \text{put}_{bmaximum} [2, 5] 4$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Right } (2, \text{get}_{bmaximum} [5])) 4 = \text{Right } (2, 4) \}$   
 $4 : 2 : \text{put}_{bmaximum} [5] 4$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Right } (5, \text{get}_{bmaximum} [])) 4 = \text{Right } (4, -\infty) \}$   
 $4 : 2 : 4 : \text{put}_{bmaximum} [] (-\infty)$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Left } ()) (-\infty) = \text{Left } () \}$   
 $4 : 2 : 4 : []$

Also, we can change the output 9 to a bigger value such as 10 and put it back to the input  $[9, 2, 5]$ , which is shown in the following calculation.

$\text{put}_{bmaximum} [9, 2, 5] 10$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Right } (9, \text{get}_{bmaximum} [2, 5])) 10 = \text{Right } (10, 5) \}$   
 $10 : \text{put}_{bmaximum} [2, 5] 5$   
 $= \{ \text{since } \text{put}_{bmax} (\text{Right } (2, \text{get}_{bmaximum} [5])) 5 = \text{Right } (2, 5) \}$

```

1795 10 : 2 : putbmaximum [5] 5
1796 = { since putbmax (Right (5, getbmaximum [])) 5 = Right (5, -∞) }
1797 10 : 2 : 5 : putbmaximum [] (-∞)
1798 = { since putbmax (Left ()) (-∞) = Left () }
1799 10 : 2 : 5 : []
1800
1801

```

### 3.2 Example of *bmap'*

The following defines a bidirectional version for  $map (*2) : [Int] \rightarrow [Int]$  where the result list only contains even numbers.

```

1806 bdouble : [Int] ↔ [Int]
1807 bdouble = bmap' bdouble
1808 where bdouble : Int ↔ Int
1809         bdouble = CLens (*2) (λ _ v' → div v' 2) ctrue (λ _ b → mod b 2 = 0)
1810
1811

```

### 3.3 Example of *bmapl*

With the help of *bmapl*, we are able to handle any constraint on adjacent elements of a list, such as partial order relations. Consider a unidirectional computation  $map (\lambda x \rightarrow mod\ x\ 10) \circ sort : [Int] \rightarrow [Int]$ , which sorts the list first and then applies the modulo 10 operation on each element. The *sort* can be bidirectionalized as follows using some auxiliary functions from the *Data.List* module of Haskell:

```

1819 bsort : [Int] ↔ [Int]
1820 bsort = CLens sort p ctrue (λ t as → (∀ 1 < i ≤ |as|. asi-1 ≤ asi) ∧ eqlength t as)
1821 where
1822   p s v = let positions = map fst $ sortOn snd (zip [0..] s) in
1823           map snd $ sortOn fst (zip positions v)
1824

```

Thus, the backward transformation of  $map (\lambda x \rightarrow mod\ x\ 10)$  should produce a sorted list. With the help of *bmapl*, we can write a bidirectional version for  $map (\lambda x \rightarrow mod\ x\ 10)$  as follows.

```

1828 bmapl (≤) ctrue (-∞) bmod10 : [S] ↔ [V]
1829 where
1830   bmod10 : (a : S) ⇒ (S ↔ V)
1831   bmod10 a = CLens (λ x → mod x 10) p (λ _ a' → a ≤ a') ctrue
1832   where p x y = if mod x 10 = y then go x else go y
1833           go x = if x > a then x else go (x + 10)
1834

```

Now we have  $bsort; bmapl (-∞) bmod10 : [Int] \leftrightarrow [Int]$  which synchronizes a list with the result list of each element modulo 10 after it is sorted.

```

1835
1836
1837
1838
1839
1840

```



### 3.4 Example of $\text{bfoldl}_{\text{init}}$

In this example, we give a bidirectional version of the computation of prefix sums. An intuitive implementation of prefix sums is  $\text{map} (\text{foldl} (+) 0) \circ \text{inits}$ . With the help of  $\text{bmapl}$  and  $\text{bfoldl}_{\text{init}}$ , we can easily bidirectionalize it as

$$\text{bprefixSum} = \text{binits}; \text{bmapl} (\lambda s s' \rightarrow \text{init } s' = s) \text{ctrue []} (\text{bfoldl}_{\text{init}} \text{ctrue } 0 \text{badd}) : [\text{Int}] \leftrightarrow [\text{Int}]$$

where the  $\text{badd}$  is defined as

$$\begin{aligned} \text{badd} &: (b : \text{Int}) \Rightarrow (\text{Either } () (\text{Int}, \text{Int}) \leftrightarrow \text{Int}) \\ \text{badd } b &= \text{CLens } g \text{ } p (\lambda \_ t' \rightarrow t' = \text{Right } (\_, b)) \text{ctrue} \\ \textbf{where } g (\text{Left } ()) &= 0 \\ g (\text{Right } (x, y)) &= x + y \\ p \_ s &= \text{Right } (s - b, b) \end{aligned}$$

This implementation of bidirectional prefix sum fits our intuition that a list of integers is isomorphic to its prefix sums. For example,  $\text{get}_{\text{bprefixSum}} [1, 2, 3]$  yields  $[1, 3, 6]$ , and  $\text{put}_{\text{bprefixSum}} [1, 2, 3] [4, 6, 8]$  yields  $[4, 2, 2]$  regardless of what the original list is.

This is a good example showing the expressive power of contract lenses in writing specifications solving bidirectional programming problems: we can decompose a complex bidirectional problem into subproblems and solve them independently. With the help of contracts (source and view conditions), they can be composed safely to solve the original problem.

### 3.5 Example of $\text{bscanl}$

Consider that we want to synchronize a list of integers with its prefix products. The forward transformation is characterized by  $\text{prefixProd} = \text{scanl}' (*) 1 : [\text{Int}] \rightarrow [\text{Int}]$ . Note that there is a constraint on the adjacent elements of the view list: the preceding element divides the following element. This constraint can be expressed with the help of  $\text{bscanl}$ .

$$\begin{aligned} \text{bprefixProd} &: ([\text{Int}] \leftrightarrow [\text{Int}]) \\ \text{bprefixProd} &= \text{bscanl} (\lambda b b' \rightarrow \text{mod } b' b = 0) 1 \text{bmul} \\ \textbf{where} \\ \text{bmul} &: (b : \text{Int}) \Rightarrow (\text{Either } () (\text{Int}, \text{Int}) \leftrightarrow \text{Int}) \\ \text{bmul } b &= \text{CLens } g \text{ } p (\lambda \_ t' \rightarrow t' = \text{Right } (\_, b)) (\lambda \_ b' \rightarrow \text{mod } b' b = 0) \\ \textbf{where } g (\text{Left } ()) &= 1 \\ g (\text{Right } (x, y)) &= x * y \\ p \_ b' &= \text{Right } (\text{div } b' b, b) \end{aligned}$$

### 3.6 Example of Bidirectional Scan Lemma

We give a simple example which makes use of the [BIDIRECTIONAL SCAN LEMMA](#) to derive an efficient bidirectional program from an inefficient one. Recall the  $\text{bprefixSum}$

defined in Appendix 3.4 for calculating the prefix sums of a list. It has time complexity  $O(n^2)$ . Applying the **BIDIRECTIONAL SCAN LEMMA** to it, we can derive *bscanl true 0 badd*, which has time complexity  $O(n)$  in both forward and backward transformations.

1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932